OS/390

MVS Programming: Assembler Services Guide

GC28-1762-06

OS/390

MVS Programming: Assembler Services Guide

GC28-1762-06

Note

Before using this information and the product it supports, be sure to read the general information under Appendix C, “Notices” on page C-1.

Seventh Edition, September 1999

This is a major revision of GC28-1762-05.

This edition applies to Version 2 Release 8 of OS/390 (5647-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

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

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+914+432-9405
FAX (Other Countries):
Your International Access Code +1+914+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.ibm.com/s390/os390/

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

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

© Copyright International Business Machines Corporation 1988, 1999. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book xvii
Who Should Use This Book xvii
How to Use This Book xvii
Where to Find More Information xviii

Summary of Changes xix

Chapter 1. Introduction 1-1

Chapter 2. Linkage Conventions 2-1
Saving the Calling Program's Registers 2-2
Caller-Provided Save Area 2-2
Linkage Convention for Floating Point Registers 2-2
Linkage Convention for the Floating Point Control Register 2-2
System-Provided Linkage Stack 2-3
Using the Linkage Stack 2-3
Example of Using the Linkage Stack 2-3
Using a Caller-Provided Save Area 2-4
Example of Using the Caller-Provided Save Area 2-5
Establishing a Base Register 2-6
Linkage Procedures for Primary Mode Programs 2-6
Primary Mode Programs Receiving Control 2-6
Primary Mode Programs Returning Control 2-7
Primary Mode Programs Calling Another Program 2-8
Linkage Procedures for AR Mode Programs 2-8
AR Mode Programs Receiving Control 2-8
AR Mode Programs Returning Control 2-9
AR Mode Programs Calling Another Program 2-9
Conventions for Passing Information Through a Parameter List 2-9
Program in Primary Mode 2-9
Programs in AR Mode 2-10

Chapter 3. Subtask Creation and Control 3-1
Creating the Task 3-1
Priorities 3-1
Address Space Priority 3-2
Task Priority 3-2
Subtask Priority 3-2
Assigning and Changing Priority 3-3
Stopping and Restarting a Subtask (STATUS Macro) 3-3
Task and Subtask Communications 3-3

Chapter 4. Program Management 4-1
Residency and Addressing Mode of Programs 4-1
Residency Mode Definitions 4-2
Addressing Mode Definitions 4-2
Linkage Considerations 4-2
Floating Point Considerations 4-3
Passing Control Between Programs with the Same AMODE 4-3
Passing Control Between Programs with Different AMODEs 4-3

© Copyright IBM Corp. 1988, 1999

Passing Control Between Programs with All Registers Intact 4-4
Load Module Structure Types 4-6
Simple Structure 4-7
Dynamic Structure 4-7
Load Module Execution 4-7
Passing Control in a Simple Structure 4-7
Passing Control without Return 4-8
Passing Control with Return 4-10
Passing Control in a Dynamic Structure 4-16
Bringing the Load Module into Virtual Storage 4-16
Passing Control with Return 4-22
Passing Control without Return 4-26
APF-authorized Programs and Libraries 4-29
Additional Entry Points 4-29
Entry Point and Calling Sequence Identifiers as Debugging Aids 4-30
Retrieving Information About Loaded Modules 4-30
Using the CSVINFO Macro 4-30
Coding a MIPR for the CSVINFO Macro 4-33
Using CSVRTLS to Request Run-Time Library Services (RTLS) 4-34

Chapter 5. Understanding 31-Bit Addressing 5-1
Virtual Storage 5-1
Addressing Mode and Residency Mode 5-1
Requirements for Execution in 31-Bit Addressing Mode 5-4
Rules and Conventions for 31-Bit Addressing 5-4
Mode Sensitive Instructions 5-4
Branching Instructions 5-5
Use of 31-Bit Addressing 5-6
Planning for 31-Bit Addressing 5-6
Converting Existing Programs 5-7
Writing New Programs That Use 31-Bit Addressing 5-9
Writing Programs for MVS/370 and MVS Systems with 31-Bit Addressing . 5-11
Addressing Mode and Residency Mode 5-12
Addressing Mode - AMODE 5-12
Residency Mode - RMODE 5-13
AMODE and RMODE Combinations 5-13
AMODE and RMODE Combinations at Execution Time 5-13
Determining the AMODE and RMODE of a Load Module 5-14
Assembler H Support of AMODE and RMODE 5-14
DFP Linkage Editor Support of AMODE and RMODE 5-15
DFP Loader Support for AMODE and RMODE 5-19
MVS Support of AMODE and RMODE 5-20
How to Change Addressing Mode 5-22
Establishing Linkage 5-23
Using the BASSM and BSM Instructions 5-26
Using Pointer-Defined Linkage 5-28
Using Supervisor-Assisted Linkage 5-31
Linkage Assist Routines 5-33
Using Capping - Linkage Using a Prologue and Epilogue 5-37
Performing I/O in 31-Bit Addressing Mode 5-38
Using the EXCP Macro 5-39
Using EXCPVR 5-39
Understanding the Use of Central Storage 5-49
Central Storage Considerations for User Programs 5-50

Chapter 6. Resource Control 6-1
Synchronizing Tasks (WAIT, POST, and EVENTS Macros) 6-2
Synchronizing Tasks (Pause, Release, and Transfer) 6-4
Pause Elements and Pause Element Tokens 6-4
Using the Services 6-5
Serializing Access to Resources (ENQ and DEQ Macros) 6-9
Naming the Resource 6-10
Defining the Scope of a Resource 6-11
Requesting Exclusive or Shared Control 6-12
Limiting Concurrent Requests for Resources 6-12
Processing the Requests 6-12
Serializing Access to Resources through the RESERVE Macro 6-17
Collecting Information about Resources and Their Requestors (GQSCAN Macro) 6-17
How GQSCAN Returns Resource Information 6-18

Chapter 7. Program Interruption Services 7-1
Specifying User Exit Routines 7-1
Using the SPIE Macro 7-2
Using the ESPIE Macro 7-3
Environment Upon Entry to User's Exit Routine 7-4
Functions Performed in User Exit Routines 7-5

Chapter 8. Providing Recovery 8-1
Understanding General Recovery Concepts 8-2
Deciding Whether to Provide Recovery 8-3
Understanding Errors in MVS 8-4
Understanding Recovery Routine States 8-5
Understanding the Various Routines in a Recovery Environment 8-6
Choosing the Appropriate Recovery Routine 8-7
Understanding Recovery Routine Options 8-9
Understanding How Routines in a Recovery Environment Interact 8-9
Writing Recovery Routines 8-11
Understanding What Recovery Routines Do 8-12
Understanding the Means of Communication 8-19
Special Considerations for ESTAE-Type Recovery Routines 8-26
Understanding the Recovery Environment 8-30
Register Contents 8-31
Other Environmental Factors in Recovery 8-37
Understanding Recovery through a Coded Example 8-42
Understanding Advanced Recovery Topics 8-45
Invoking RTM (ABEND Macro) 8-45
Providing Multiple Recovery Routines 8-46
Providing Recovery for Recovery Routines 8-47
Providing Recovery for Multitasking Programs 8-47
Using STAE/STAI Routines. 8-48

Chapter 9. Dumping Virtual Storage (ABEND, SNAPX, SNAP, and IEATDUMP Macros) 9-1
ABEND Dumps 9-2
Obtaining a Symptom Dump 9-2
Suppressing Dumps That Duplicate Previous Dumps 9-2
SNAP Dumps 9-9
Finding Information in a SNAP Dump 9-9

Obtaining a Summary Dump for an ABEND or SNAP Dump 9-9
Transaction Dumps 9-10

Chapter 10. Reporting Symptom Records (SYMRBLD and SYMREC Macros) 10-1
Writing Symptom Records to Logrec Data Set 10-1
The Format of the Symptom Record 10-2
Symptom Strings — SDB Format 10-3
Building a Symptom Record Using the SYMRBLD Macro 10-3
Building a Symptom Record Using the ADSR and SYMREC Macros 10-4
Programming Notes for Section 1 10-4
Programming Notes for Section 2 10-5
Programming Notes for Section 2.1 10-6
Programming Notes for Section 3 10-7
Programming Notes for Section 4 10-8
Programming Notes for Section 5 10-8

Chapter 11. Virtual Storage Management 11-1
Explicit Requests for Virtual Storage 11-1
Obtaining Storage Through the GETMAIN Macro 11-2
Obtaining Storage Through the STORAGE Macro 11-4
Using the CPOOL Macro 11-6
Subpool Handling 11-6
Implicit Requests for Virtual Storage 11-11
Reenterable Load Modules 11-11
Reenterable Macros 11-11
Non-reenterable Load Modules 11-13
Freeing of Virtual Storage 11-13

Chapter 12. Callable Cell Pool Services 12-1
Comparison of CPOOL Macro and Callable Cell Pool Services 12-1
Storage Considerations 12-2
Link-editing Callable Cell Pool Services 12-4
Using Callable Cell Pool Services 12-4
Handling Return Codes 12-6
Callable Cell Pool Services Coding Example 12-6

Chapter 13. Data-in-Virtual 13-1
When to Use Data-in-Virtual 13-2
Factors Affecting Performance 13-2
Creating a Linear Data Set 13-3
Using the Services of Data-in-Virtual 13-4
Identify 13-4
Access 13-4
Map 13-4
Save, Savelist, and Reset 13-6
Unmap 13-6
Unaccess 13-7
Unidentify 13-7
The IDENTIFY Service 13-7
The ACCESS Service 13-8
The MAP Service 13-10
The SAVE Service 13-15
The SAVELIST Service 13-17

The RESET Service 13-18
Effect of RETAIN mode on RESET 13-18
The UNMAP Service 13-20
The UNACCESS and UNIDENTIFY Services 13-21
Sharing Data in an Object 13-21
Miscellaneous Restrictions for Using Data-in-Virtual 13-22
DIV Macro Programming Examples 13-22
General Program Description 13-22
Data-in-Virtual Sample Program Code 13-23
Executing the Program 13-29

Chapter 14. Using Access Registers 14-1
Access Lists 14-3
Types of Access Lists 14-3
Writing Programs in AR Mode 14-5
Coding Instructions in AR Mode 14-6
Manipulating the Contents of ARs 14-7
Loading an ALET into an AR 14-8
Loading the Value of Zero into an AR 14-8
The ALESERV Macro 14-9
Adding an Entry to an Access List 14-9
Deleting an Entry from an Access List 14-10
Issuing MVS Macros in AR Mode 14-10
Example of Using SYSSTATE 14-10
Using X-Macros 14-11
Formatting and Displaying AR Information 14-11

Chapter 15. Data Spaces and Hiperspaces 15-1
What are Data Spaces and Hiperspaces? 15-1
What Can a Program Do With a Data Space or a Hiperspace? 15-2
How Does a Program Obtain a Data Space and a Hiperspace? 15-2
How Does a Program Move Data into a Data Space or Hiperspace? . . . 15-2
Who Owns a Data Space or Hiperspace? 15-2
Can an Installation Limit the Use of Data Spaces and Hiperspaces? 15-3
How Does a Program Manage the Storage in a Data Space or Hiperspace? 15-3
Differences Between Data Spaces and Hiperspaces 15-4
Comparing Data Space and Hiperspace Use of Physical Storage 15-5
Which One Should Your Program Use? 15-6
An Example of Using a Data Space 15-6
An Example of Using a Hiperspace 15-6
Creating and Using Data Spaces 15-7
Manipulating Data in a Data Space 15-7
Rules for Creating, Deleting, and Managing Data Spaces 15-7
Creating a Data Space 15-8
Establishing Addressability to a Data Space 15-12
Examples of Moving Data into and out of a Data Space 15-12
Extending the Current Size of a Data Space 15-14
Releasing Data Space Storage 15-16
Paging Data Space Storage Areas into and out of Central Storage 15-16
Deleting a Data Space 15-17
Using Callable Cell Pool Services to Manage Data Space Areas 15-17
Sharing Data Spaces among Problem-State Programs with PSW Keys 8 - F 15-19

Sharing Data Spaces through the PASN-AL 15-21
Example of Mapping a Data-in-Virtual Object to a Data Space 15-22
Using Data Spaces Efficiently 15-23
Example of Creating, Using, and Deleting a Data Space 15-23
Dumping Storage in a Data Space 15-25
Using Checkpoint/Restart 15-25
Creating and Using Hiperspaces 15-25
Standard Hiperspaces 15-26
Creating a Hiperspace 15-28
Transferring Data To and From Hiperspaces 15-29
Extending the Current Size of a Hiperspace 15-34
Releasing Hiperspace Storage 15-34
Deleting a Hiperspace 15-35
Example of Creating a Standard Hiperspace and Using It 15-35
Using Data-in-Virtual with Hiperspaces 15-37
Using Checkpoint/Restart 15-41

Chapter 16. Window Services 16-1
Data Objects 16-1
Permanent 16-1
Temporary Data Objects 16-1
Structure of a Data Object 16-1
What Does Window Services Provide? 16-2
The Ways That Window Services Can Map an Object 16-3
Access to Permanent Data Objects 16-6
Access to Temporary Data Objects 16-7
Using Window Services 16-8
Obtaining Access to a Data Object 16-9
Defining a View of a Data Object 16-11
Defining the Expected Reference Pattern 16-12
Defining Multiple Views of an Object 16-14
Saving Interim Changes to a Permanent Data Object 16-15
Updating a Temporary Data Object 16-16
Refreshing Changed Data 16-16
Updating a Permanent Object on DASD 16-17
Changing a View in a Window 16-17
Terminating Access to a Data Object 16-19
Link-editing Callable Window Services 16-19
Window Services Coding Example 16-20

Chapter 17. Sharing Application Data (Name/Token Callable Services) . 17-1
Understanding Name/Token Pairs and Levels 17-1
Name/Token Pairs 17-1
Levels for Name/Token Pairs 17-2
Determining What Your Program Can Do with Name/Token Pairs 17-3
Deciding What Name/Token Level You Need 17-3
Task-Level Name/Token Pair 17-3
Home-Level Name/Token Pair 17-4
Owning and Deleting Name/Token Pairs 17-6
Using Checkpoint/Restart with Name/Token Pairs 17-6
Link-Editing Name/Token Services 17-6

Chapter 18. Processor Storage Management 18-1
Freeing Virtual Storage 18-2

Releasing Storage 18-2
Protecting a Range of Virtual Storage Pages 18-3
Loading/Paging Out Virtual Storage Areas 18-3
Virtual Subarea List (VSL) 18-4
Page Service List (PSL) 18-5
Defining the Reference Pattern (REFPAT) 18-5
How Does the System Handle the Data in an Array? 18-6
Using the REFPAT Macro 18-10
Examples of Using REFPAT to Define a Reference Pattern 18-14
Removing the Definition of the Reference Pattern 18-15

Chapter 19. Sharing Data in Virtual Storage (IARVSERV Macro) 19-1
Understanding the Concepts of Sharing Data with IARVSERV 19-2
Storage You Can Use with IARVSERV 19-2
Obtaining Storage for the Source and Target 19-3
Defining Storage for Sharing Data and Access 19-3
Changing Storage Access 19-4
How to Share and Unshare Data 19-6
Accessing Data in a Sharing Group 19-7
Example of Sharing Storage with IARVSERV 19-7
Use with Data-in-Virtual (DIV Macro) 19-8
Diagnosing Problems with Shared Data 19-9
Converting a Central to Virtual Storage Address (IARR2V Macro) 19-9

Chapter 20. Timing and Communication 20-1
Checking for Timer Synchronization 20-1
Obtaining Time of Day and Date 20-1
Converting Between Time of Day and Date and TOD Clock Formats 20-2
Interval Timing 20-2
Obtaining Accumulated Processor Time 20-4
Writing and Deleting Messages (WTO, WTOR, DOM, and WTL) 20-5
Routing the Message 20-6
Writing a Multiple-Line Message 20-10
Embedding Label Lines in a Multiple-Line Message 20-10
Communicating in a Sysplex Environment 20-10
Writing to the Programmer 20-10
Writing to the System Log 20-11
Deleting Messages Already Written 20-11
Retrieving Console Information (CONVCON Macro) 20-12
Determining the Name or ID of a Console 20-12
Validating a Console Name or ID and Checking if a Console Is Active . . . 20-13
Validating a Console Area ID 20-14

Chapter 21. Translating Messages 21-1
Allocating Data Sets for an Application 21-3
Creating Install Message Files 21-4
Creating a Version Record 21-4
Creating Message Skeletons 21-4
Message Skeleton Format 21-5
Message Text in a Skeleton 21-6
Validating Message Skeletons 21-7
Allocating Storage for Validation Run-Time Message Files 21-8
Compiling Message Files 21-9
Checking the Message Compiler Return Codes 21-12

Updating the System Run-Time Message Files 21-13
Using MMS Translation Services in an Application 21-13
Determining which Languages are Available (QRYLANG Macro) 21-14
Retrieving Translated Messages (TRANMSG Macro) 21-14
Example of Displaying Messages 21-16
Using Message Parameter Blocks for New Messages (BLDMPB and UPDTMPB Macros) 21-17
Support for Additional Languages 21-18
Example of an Application that Uses MMS Translation Services 21-19

Chapter 22. Data Compression and Expansion Services 22-1
Services Provided by CSRCESRV 22-1
Running under an MVS/ESA System 22-2
Running under an MVS/XA System 22-3
Services Provided by CSRCMPSC 22-4
Compression and Expansion Dictionaries 22-5
Building the CSRYCMPS Area 22-5
Determining if the CSRCMPSC Macro Can Be Issued on a System 22-8
Compression Processing 22-9
Expansion Processing 22-10
Dictionary Entries 22-10

Chapter 23. Accessing Unit Control Blocks (UCBs) 23-1
Detecting I/O Configuration Changes 23-1
Scanning UCBs 23-2
Obtaining UCB Information for a Specified Device 23-3
Obtaining Eligible Device Table Information 23-4
Using the EDTINFO Macro 23-4

Chapter 24. The Internal Reader 24-1
Setting Up and Using an Internal Reader 24-1
Allocating the Internal Reader Data Set 24-2
Opening the Internal Reader Data Set 24-2
Sending Job Output to the Internal Reader 24-3
Closing the Internal Reader Data Set 24-3

Chapter 25. Using the Symbol Substitution Service 25-1
What Are Symbols? 25-1
Types of Symbols 25-1
Examples of User Symbols 25-2
Calling the ASASYMBM Service 25-3
Setting Up the ASASYMBP Mapping Macro 25-3
Providing a Symbol Table to ASASYMBM 25-4
Using Symbols in Programs 25-10

Chapter 26. Using System Logger Services 26-1
What is System Logger? 26-1
The Log Stream 26-2
The System Logger Configuration 26-4
The System Logger Component 26-6
The LOGR Couple Data Set (LOGR Policy) 26-7
Log Data on The Coupling Facility 26-8
Log Data on DASD Log Data Sets 26-8
Duplexing Log Data 26-9

Overview of System Logger Services 26-10
Summary of System Logger Services 26-10
Define Authorization to System Logger Resources 26-11
Synchronous and Asynchronous Processing 26-11
Coding a System Logger Complete Exit for IXGBRWSE, IXGWRITE, and IXGDELET 26-13
How System Logger Handles Gaps in the Log Stream 26-15
Dumping on Data Loss (804–type) Conditions 26-16
Using the System Logger Answer Area (ANSAREA parameter) 26-18
Using ENF Event Code 48 in System Logger Applications 26-19
IXGINVNT: Managing the LOGR Policy 26-20
Defining a Model Log Stream in the LOGR Couple Data Set 26-21
Defining a Log Stream as DASD-Only 26-22
Upgrading a Log Stream From DASD-Only to Coupling Facility 26-22
IXGCONN: Connecting to and Disconnecting From a Log Stream 26-22
Examples of Ways to Connect to the Log Stream 26-23
How System Logger Allocates Structure Space For a New Log Stream at Connection Time 26-24
Connect Process and Staging Data Sets 26-24
Requesting Authorization to the Log Stream for an Application 26-24
Connecting as a Resource Manager 26-25
Requesting a Write or Import Connection - IMPORTCONNECT parameter 26-26
Specifying User Data for a Log Stream 26-27
Using ENF Event 48 When a Connect Request is Rejected 26-27
System Logger Processing at Disconnection 26-27
Coding a Resource Manager Exit for IXGCONN 26-28
IXGWRITE: Writing to a Log Stream 26-32
The Log Block Buffer 26-32
Ensuring Chronological Sequence of Log Blocks 26-32
When is Data Committed to the Log Stream? 26-33
When the Log Stream Coupling Facility Storage Limit Is Reached 26-33
When the Staging Data Set Storage Limit is Reached 26-34
When the Staging Data Set is Formatting 26-34
IXGBRWSE: Browsing/Reading a Log Stream 26-34
IXGBRWSE Terminology 26-34
IXGBRWSE Requests 26-35
Browsing Both Active and Inactive Data 26-35
Browsing for a Log Block by Time Stamp 26-36
Using IXGBRWSE and IXGWRITE 26-37
Using IXGBRWSE and IXGDELET Requests Together 26-37
IXGDELET: Deleting Log Blocks from a Log Stream 26-38
Using the BLOCKS parameter 26-38
Delete Requests and Resource Manager Exit Processing 26-39
IXGIMPRT: Import Log Blocks 26-39
Making Sure Log Blocks are Imported in Sequence - Understanding Log Block Identifiers 26-40
Making Sure Log Data is Safe to Import 26-41
IXGQUERY: Get Information About a Log Stream 26-41
The Safe Import Point: Using IXGQUERY and IXGIMPRT Together 26-41
The Coupling Facility List Structure Version Number 26-44
IXGOFFLD: Initiate Offload to DASD Log Data Sets 26-45
Managing a Target Log Stream: Using IXGIMPRT, IXGOFFLD, and IXGQUERY Together 26-46
IXGUPDAT: Modify Log Stream Control Information 26-46

Rebuilds and IXGUPDAT processing . . . 26-47
Setting Up the System Logger Configuration . . . 26-47
Writing an ENF Event 48 Listen Exit . . . 26-47
Reading Data From Log Streams in Data Set Format . . . 26-49
Is My Application Eligible for the LOGR Subsystem? . . . 26-49
Using the LOGR Subsystem . . . 26-51
JCL for the LOGR Subsystem . . . 26-51
When Things Go Wrong - Recovery Scenarios for System Logger . . . 26-53
When a System Logger Application Fails . . . 26-53
When an MVS System or Sysplex Fails . . . 26-54
Recovery Performed for DASD-Only Log Streams . . . 26-54
When the System Logger Address Space Fails . . . 26-54
When the Coupling Facility Structure Fails . . . 26-55
When the Coupling Facility Space for a Log Stream Becomes Full . . . 26-57
When a Staging Data Set Becomes Full . . . 26-57
When a Log Stream is Damaged . . . 26-58
When DASD Log Data Set Space Fills . . . 26-58
When Unrecoverable DASD I/O Errors Occur . . . 26-59
When A Resource Manager Fails . . . 26-60

Appendix A. Using the Unit Verification Service . . . A-1
Functions of Unit Verification . . . A-1
Check Groups - Function Code 0 . . . A-1
Check Units - Function Code 1 . . . A-1
Return Unit Name - Function Code 2 . . . A-2
Return Unit Control Block (UCB) Addresses - Function Code 3 . . . A-2
Return Group ID - Function Code 4 . . . A-2
Indicate Unit Name is a Look-up Value - Function Code 5 . . . A-2
Return Look-up Value - Function Code 6 . . . A-2
Convert Device Type to Look-up Value - Function Code 7 . . . A-2
Return Attributes - Function Code 8 . . . A-2
Specify Subpool for Returned Storage - Function Code 10 . . . A-2
Return Unit Names for a Device Class - Function Code 11 . . . A-3

Appendix B. Using the Virtual Fetch Service . . . B-1
Functions of Virtual Fetch . . . B-1
Considerations . . . B-2
Programming Conventions for Using Virtual Fetch . . . B-4
Requesting Dumps When Using Virtual Fetch . . . B-4
Return Codes from the BUILD Request for Virtual Fetch . . . B-5
Return Codes from the FIND Request for Virtual Fetch . . . B-5
Return Codes from the GET Request for Virtual Fetch . . . B-5

Appendix C. Notices . . . C-1
Programming Interface Information . . . C-2
Trademarks . . . C-2

Index . . . X-1

Figures

2-1. Format of the Save Area 2-4
2-2. Primary Mode Parameter List 2-10
2-3. AR Mode Parameter List 2-11
3-1. Levels of Tasks in a Job Step 3-4
4-1. Assembler Definition of AMODE/RMODE 4-1
4-2. Example of Addressing Mode Switch 4-4
4-3. Characteristics of Load Modules 4-7
4-4. Passing Control in a Simple Structure 4-9
4-5. Passing Control With a Parameter List 4-9
4-6. Passing Control With Return 4-11
4-7. Passing Control With CALL 4-11
4-8. Test for Normal Return 4-13
4-9. Return Code Test Using Branching Table 4-13
4-10. Establishing a Return Code 4-14
4-11. Using the RETURN Macro 4-15
4-12. RETURN Macro with Flag 4-15
4-13. Search for Module, EP or EPLOC Parameter With DCB=0 or DCB Parameter Omitted 4-18
4-14. Search for Module, EP or EPLOC Parameters With DCB Parameter Specifying Private Library 4-19
4-15. Search for Module Using DE Parameter 4-21
4-16. Use of the LINK Macro with the Job or Link Library 4-24
4-17. Use of the LINK Macro with a Private Library 4-24
4-18. Use of the BLDL Macro 4-24
4-19. The LINK Macro with a DE Parameter 4-24
4-20. Misusing Control Program Facilities Causes Unpredictable Results 4-28
4-21. Processing Flow for the CSVINFO Macro and the Caller's MIPR . . 4-32
4-22. CSVINFO Recovery 4-33
5-1. Two Gigabyte Virtual Storage Map 5-2
5-2. Maintaining Correct Interfaces to Modules that Change to AMODE 31 5-7
5-3. Establishing Correct Interfaces to Modules That Move Above 16 Megabytes 5-8
5-4. AMODE and RMODE Combinations 5-14
5-5. AMODE and RMODE Processing by the Linkage Editor 5-17
5-6. AMODE and RMODE Processing by the Loader 5-20
5-7. Mode Switching to Retrieve Data from Above 16 Megabytes 5-23
5-8. Linkage Between Modules with Different AMODEs and RMODEs . 5-25
5-9. BRANCH and SAVE and Set Mode Description 5-26
5-10. Branch and Set Mode Description 5-27
5-11. Using BASSM and BSM 5-28
5-12. Example of Pointer-Defined Linkage 5-30
5-13. Example of Supervisor-Assisted Linkage 5-32
5-14. Example of a Linkage Assist Routine 5-34
5-15. Cap for an AMODE 24 Module 5-38
5-16. Performing I/O While Residing Above 16 Megabytes 5-41
| 6-1. Task Synchronization Techniques 6-1
6-2. Event Control Block (ECB) 6-2
6-3. Using LINKAGE=SYSTEM on the WAIT and POST Macros 6-3
| 6-4. Pause Element (PE) and Event Control Block (ECB) 6-4
| 6-5. Pause and Release Example 6-6

© Copyright IBM Corp. 1988, 1999

6-6. Release and Pause Example 6-7
6-7. Transfer without Pause Example 6-9
6-8. ENQ Macro Processing 6-13
6-9. Interlock Condition 6-16
6-10. Two Requests For Two Resources 6-16
6-11. One Request For Two Resources 6-16
6-12. GQSCAN Results with a Scope of STEP, SYSTEM, SYSTEMS, or ALL 6-19
6-13. Work Area Contents for GQSCAN with a Scope of STEP, SYSTEM, SYSTEMS, or ALL 6-20
7-1. Using the SPIE Macro 7-3
8-1. Summary of Recovery Routine States 8-9
8-2. Mainline Routine with One Recovery Routine 8-10
8-3. Mainline Routine with Several Recovery Routines 8-11
8-4. Contents of GPR 0 on Entry to a Retry Routine 8-18
8-5. Key Fields in the SDWA 8-23
8-6. Restoring Quiesced Restorable I/O Operations 8-29
8-7. Where to Find Register Content Information 8-32
8-8. Register Contents—ESTAE-Type Recovery Routine With an SDWA 8-33
8-9. Register Contents—ESTAE-Type Recovery Routine Without an SDWA 8-34
8-10. Register Contents—Retry from an ESTAE-Type Recovery Routine Without an SDWA 8-35
8-11. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=NO 8-36
8-12. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=YES 8-36
8-13. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA and RETREGS=YES 8-37
8-14. Environments of ESTAE-type Recovery Routines and their Retry Routines 8-40
9-1. Reasons for Selecting the Type of Dump 9-1
11-1. Example of Using the GETMAIN Macro 11-4
11-2. Virtual Storage Control 11-7
11-3. Using the List and the Execute Forms of the DEQ Macro 11-12
12-1. Cell Pool Storage 12-3
13-1. Mapping from an Address Space 13-11
13-2. Mapping from a Data Space or Hiperspace 13-11
13-3. Multiple Mapping 13-13
14-1. Using an ALET to Identify an Address Space or a Data Space . . . 14-2
14-2. An Illustration of a DU-AL 14-4
14-3. Characteristics of DU-ALs and PASN-ALs 14-5
14-4. Using Instructions in AR Mode 14-6
14-5. Base and Index Register Addressing in AR Mode 14-7
15-1. Accessing Data in a Data Space 15-4
15-2. Accessing Data in a Hiperspace 15-5
15-3. Rules for How Problem State Programs with Key 8-F Can Use Data Spaces 15-8
15-4. Example of Specifying the Size of a Data Space 15-11
15-5. Example of Extending the Current Size of a Data Space 15-16
15-6. Example of Using Callable Cell Pool Services for Data Spaces . . 15-19
15-7. Two Problem Programs Sharing a SCOPE=SINGLE Data Space 15-20
15-8. Example of Scrolling through a Standard Hiperspace 15-27
15-9. Facts about a Non-shared Standard Hiperspace 15-28

15-10. Illustration of the HSPSERV Write and Read Operations 15-30
15-11. Example of Creating a Standard Hiperspace and Transferring Data 15-36
15-12. Example of Mapping a Data-in-Virtual Object to a Hiperspace . . 15-38
15-13. A Standard Hiperspace as a Data-in-Virtual Object 15-40
16-1. Structure of a Data Object 16-2
16-2. Mapping a Permanent Object That Has No Scroll Area 16-3
16-3. Mapping a Permanent Object That Has A Scroll Area 16-4
16-4. Mapping a Temporary Object 16-4
16-5. Mapping an Object To Multiple Windows 16-5
16-6. Mapping Multiple Objects 16-6
17-1. Using the Name and the Token 17-1
17-2. Summary of What Programs Do with Name/Token Pairs 17-3
17-3. Using the Task Level in a Single Address Space 17-4
17-4. Using Home-Level and Task-Level Name/Token Pairs 17-5
18-1. Releasing Virtual Storage 18-3
18-2. Example of using REFPAT with a Large Array 18-7
18-3. Illustration of a Reference Pattern with a Gap 18-8
18-4. Illustration of Forward Direction in a Reference Pattern 18-10
18-5. Illustration of Backward Direction in a Reference Pattern 18-11
18-6. Two Typical Reference Patterns 18-11
19-1. Data Sharing with IARVSERV 19-2
19-2. Allowed Source/Target View Combinations for Share 19-5
19-3. Sharing Storage with IARVSERV 19-8
20-1. Interval Processing 20-4
20-2. Characters Printed or Displayed on an MCS Console 20-5
20-3. Descriptor Code Indicators 20-8
20-4. Writing to the Operator 20-8
20-5. Writing to the Operator With a Reply 20-9
21-1. Preparing Messages for Translation 21-3
21-2. Format of Version Record Fields 21-4
21-3. Version Record Example 21-4
21-4. Message Skeleton Fields 21-5
21-5. Sample job to invoke IDCAMS to obtain a data set for the run-time message files 21-9
21-6. Using JCL to Invoke the Compiler with a single PDS as input 21-9
21-7. Using JCL to Invoke the Compiler with a concatenation of partitioned Data Sets as input 21-10
21-8. Using a TSO/E CLIST to Invoke the Compiler with a single PDS input 21-10
21-9. Using a TSO/E CLIST to Invoke the Compiler with a concatenation of partitioned Data Set as input 21-10
21-10. Using a REXX exec to Invoke the Compiler with a single PDS as input 21-11
21-11. Using a REXX exec to Invoke the Compiler with a concatenation of partitioned Data Sets as input 21-11
21-12. Using the TRANMSG Macro 21-16
21-13. Languages Available to MVS Message Service 21-18
22-1. Summary of Data Compression and Expansion Services 22-2
22-2. Testing the Level of the MVS System at Execution Time 22-3
25-1. Contiguous Symbol Table 25-8
25-2. Non-contiguous Symbol Table 25-9
26-1. System Logger Log Stream 26-2
26-2. Log Stream Data on the Coupling Facility and DASD 26-3
26-3. Log Stream Data in Local Storage Buffers and DASD Log Data Sets 26-4

26-4. A Complete Coupling Facility Log Stream Configuration 26-5
26-5. A DASD-Only Configuration 26-6
26-6. Defining SAF Authorization For System Logger Resources 26-11
26-7. How IXGBRWSE Requests Handle Gaps in a Log Stream 26-15
26-8. How IXGDELET Requests Handle Gaps in a Log Stream 26-16
26-9. Issuing ENFREQ to Listen for ENF Event Code 48 26-20
26-10. Define a Log Stream as a Model and then Model a Log Stream After It 26-21
26-11. Searching for a Log Block by Time 26-37
26-12. Deleting a Range of Log Blocks 26-39
26-13. How Source and Target Log Streams Can Get Out of Sync 26-43
26-14. Log Stream SUBSYS Data Set Specification 26-51
A-1. Input Parameter List A-4
A-2. Requesting Function Code 0 (Check Groups) A-5
A-3. Requesting Function Code 1 (Check Units) A-6
A-4. Requesting Function Code 2 (Return Unit Name) A-6
A-5. Output from Function Code 2 (Return Unit Name) A-6
A-6. Requesting Function Code 3 (Return UCB Addresses) A-7
A-7. Output from Function Code 3 (Return UCB Addresses) A-7
A-8. Requesting Function Code 4 (Return Group ID) A-8
A-9. Output from Function Code 4 (Return Group ID) A-8
A-10. Requesting Function Code 5 (Indicate Unit Name is a Look-up Value) A-9
A-11. Requesting Function Code 6 (Return Look-up Value) A-9
A-12. Output from Function Code 6 (Return Look-up Value) A-10
A-13. Requesting Function Code 7 (Convert Device Type to Look-up Value) A-10
A-14. Output from Function Code 7 (Convert Device Type to Look-up Value) A-10
A-15. Requesting Function Code 8 (Return Attributes) A-11
A-16. Requesting Function Code 10 (Specify Subpool for Returned Storage) A-12
A-17. Requesting Function Code 11 (Return Unit Names for a Device Class) A-12
A-18. Output from Function Code 11 (Return Unit Names for a Device Class) A-12
A-19. Input for Function Codes 0 and 1 . . . . . A-13
A-20. Output from Function Codes 0 and 1 . . . . . A-13
A-21. Input for Function Codes 3 and 10 . . . . . A-14
A-22. Output from Function Codes 3 and 10 . . . . . A-14
A-23. Input for Function Codes 1 and 5 . . . . . A-14
A-24. Output from Function Codes 1 and 5 . . . . . A-15
B-1. Virtual Fetch Parameter List B-3
B-2. A Program Using Virtual Fetch B-6

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, *OS/390 MVS Programming: Assembler Services Reference*, provides the detailed information for coding the macros.

Some of the topics discussed in this book are also discussed in *OS/390 MVS Programming: Authorized Assembler Services Guide* and in the following other books:

- *OS/390 MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *OS/390 MVS Programming: Authorized Assembler Services Reference ENF-IXG*
- *OS/390 MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *OS/390 MVS Programming: Authorized Assembler Services Reference SET-WTO*

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 described in the following books:

- *HLASM Programmer's Guide*
- *HLASM Language Reference*

Using this book also requires you to be familiar with the operating system and the services that programs running under it can invoke.

How to Use This Book

This book is one of the set of programming books for MVS. This set describes how to write programs in assembler language or high-level languages, such as C, FORTRAN, and COBOL. For more information about the content of this set of books, see *OS/390 Information Roadmap*.

© Copyright IBM Corp. 1988, 1999

Where to Find More Information

Where necessary, this book references information in other books, using shortened versions of the book title. The following table contains a partial list of reference books. For complete titles, and order numbers of the books for all products associated with of OS/390 see *OS/390 Information Roadmap*.

Summary of Changes

Summary of Changes
for GC28-1762-06
OS/390 Version 2 Release 8

The book contains information previously presented in GC28-1762-05, which supports OS/390 Version 2 Release 6.

New Information

- Information about Parallel Access Volume (PAV) is added to **Accessing Unit Control Blocks (UCBs)**.

Support for the following APAR is included:

- OW36830

This book includes terminology, 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.

Summary of Changes
for GC28-1762-05
OS/390 Version 2 Release 6

This book contains information previously presented in *Assembler Services Guide*, GC28-1762-04, which supports OS/390 Version 2 Release 5.

New Information

- IEEE floating point support, and the handling of the Floating Point registers and Floating Point Control register are explained.

Complete support for IEEE floating point require PTFs to OS/390 Release 6 and specific releases of some software. See *OS/390 Planning for Installation* for the software requirements for IEEE floating point.

This book includes terminology, 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.

Summary of Changes
for GC28-1762-04
OS/390 Version 2 Release 5

This book contains information previously presented in GC28-1762-03, which supports OS/390 Version 2 Release 4.

New Information

This book includes terminology, 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.

© Copyright IBM Corp. 1988, 1999

Summary of Changes
for GC28-1762-03
OS/390 Version 2 Release 4

This book contains information previously presented in GC28-1762-02, which supports OS/390 Version 1 Release 3.

New Information

- Chapter 26, “Using System Logger Services” on page 26-1 contains new information about using system logger services with a DASD-only log stream application. Changes include:
  - A new field in the system logger answer area (IXGANSAA).
  - Using the new DASDONLY parameter on the IXGINVNT service to indicate if a log stream is a DASD-only one.
  - Upgrading a log stream from DASD-only to coupling facility based.
  - Connecting to and disconnecting from (IXGCONN) a DASD-only log stream.
  - DASD-only log streams and the safe import point (IXGQUERY).
  - Recovery performed for a DASD-only log stream.
  - Recovery for when DASD log data set space fills.
- Transaction dump
- CSVRTLS is added to request functions of run-time library services (RTLS).

This book includes terminology, 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.

Summary of Changes
for GC28-1762-02
OS/390 Version 1 Release 3

This book contains information previously presented in GC28-1762-01, which supports OS/390 Version 1 Release 2.

New Information

- Chapter 26, “Using System Logger Services” on page 26-1 contains new function for system logger services, including:
  - New services: IXGIMPRT, IXGQUERY, IXGOFFLD, and IXGUPDAT.
  - New complete exit for IXGBRWSE, IXGWRITE, and IXGDELET.
  - Guidance for new keywords on existing system logger services.
  - Guidance for coding a system logger resource manager.
  - New resource manager exit for IXGCONN.
  - New ENF event 48 events.
- Adds CSRPRGT1 and CSRPRGT1 to the description of how to allocate cells.
- Adds CSRPRFR1 and CSRPRFR1 to the description of how to free cells.

This book includes terminology, 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.

Summary of Changes
for GC28-1762-01
OS/390 Release 2

This book contains information previously presented in *OS/390 MVS Programming: Assembler Services Guide*, GC28-1762-00, which supports OS/390 Release 1.

Summary of Changes
for GC28-1762-00
OS/390 Release 1

This book contains information previously presented in *MVS/ESA Programming: Assembler Services Guide*, GC28-1466, which supports MVS/ESA System Product Version 5.

This book includes terminology, 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.

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.

The chapters in this book discuss the following topics:

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 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. By using the GQSCAN macro, you can obtain information about resources and their requestors.

Program Interruption Services — The system offers many services to detect and process abnormal conditions during system processing. Some conditions encountered in a program cause program interruptions or program exceptions. This topic includes how to specify user exit routines, using the SPIE or ESPIE macros, and function performed in user exit routines.

Providing Recovery — When your program encounters an error, the program might end abnormally unless you provide recovery. To recover from errors, you can write recovery routines that get control when the error occurs. These routines can attempt to correct the error and allow your program to resume normal processing. This topic explains recovery concepts and how to write recovery routines.

Dumping Virtual Storage (ABEND, SNAPX, and SNAP Macros) — If your program makes serious errors, the system terminates it. If you request it, the

© Copyright IBM Corp. 1988, 1999

system generates a dump to accompany the termination, and the resulting dump is 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.

Reporting Symptom Records (SYMRBLD and SYMREC Macros) — An application can write a symptom record for each error to the logrec data set, the online repository where MVS collects error information. The unit of information stored in the logrec data set 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. An application can build and write symptom records in the logrec data set by invoking either the SYMRBLD or SYMREC macro.

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

Sharing Application Data (Name/Token Callable Services) — Name/token callable services allow a user to share data between two programs running under

the same task, or between two or more tasks or address spaces. This topic includes understanding what a name/token pair is, descriptions of the levels of name/token pairs, ownership and deletion of the pairs, using checkpoint/restart with name/token pairs, and an example of JCL that can link-edit a reentrant program with linkage-assist routines.

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, make virtual storage pages read-only or modifiable, 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.

Sharing Data in Virtual Storage (IARVSERV Macro) — This topic describes the IARVSERV macro, which provides services that allow programs to share virtual storage in address spaces or data spaces. The topic also includes a section for the IARR2V macro, which converts a central storage address to a virtual storage address.

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/E terminal, or to the system log.

Translating Messages — 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.

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.

Accessing Unit Control Blocks (UCBs) — 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.

The Internal Reader — The internal reader facility is a logical device similar to a card reader, a tape drive, or a TSO/E terminal that allows you to submit jobs to JES. This chapter describes how to set up and use an internal reader, allocating the internal reader data set, opening and closing the internal reader data set, and sending job output to the internal reader.

Using the Symbol Substitution Service — This topic describes types of symbols you can use in application and vendor programs, and describes how to call the ASASYMBM service, which substitutes text for those symbols.

Using System Logger Services — System logger services allow an application to manage log data in a sysplex environment. This topic describes how to plan the system logger configuration, plan and set up a system logger application, and plan for recovery for system logger applications.

Appendixes — This book also contains an appendix for each of the following topics:

- **Using the Unit Verification Service**
- **Using the Virtual Fetch Service**.

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 *OS/390 MVS Programming: Extended Addressability Guide*.

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.

© Copyright IBM Corp. 1988, 1999

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 14, “Using Access Registers” on page 14-1.

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.

Linkage Convention for Floating Point Registers

With 16 Floating Point Registers (FPRs), registers 0 to 7 are volatile, and registers 8 to 15 are non-volatile. That is, if a called routine uses any of FPRs 8 to 15, it must save the caller's data in those FPRs before use and restore them before returning to the caller. The called routine can use any of FPRs 0 to 7 without saving and restoring the caller's data. If the caller wants to keep data in FPRs 0 to 7, it must save those FPRs before the call and restore them afterward.

Linkage Convention for the Floating Point Control Register

The Floating Point Control (FPC) Register is non-volatile across calls with the exception of two fields: the IEEE exception flags and the DXC, which are volatile. That is, if a called routine changes any fields in the FPC register other than the IEEE exception flags and the DXC, it must save the caller's values before making the change and restore them before returning to the caller. The called routine may change the IEEE exception flags and DXC, explicitly or by triggering exception conditions, without saving and restoring the caller's values.

Note: A program can rely on the FPC register being zero (IEEE default) ONLY when it determines that the MVS task under which it is running is not enabled to use the AFP and FPC registers.

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 2-6
- “AR Mode Programs Receiving Control” on page 2-8

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.

If you choose not to use the SAVE and RETURN macros, you can use the IHASAVER macro to map the fields in the standard save area.

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 2-5 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 2-9.
- 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 2-9.
- 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.”
- 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 2-8.

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.

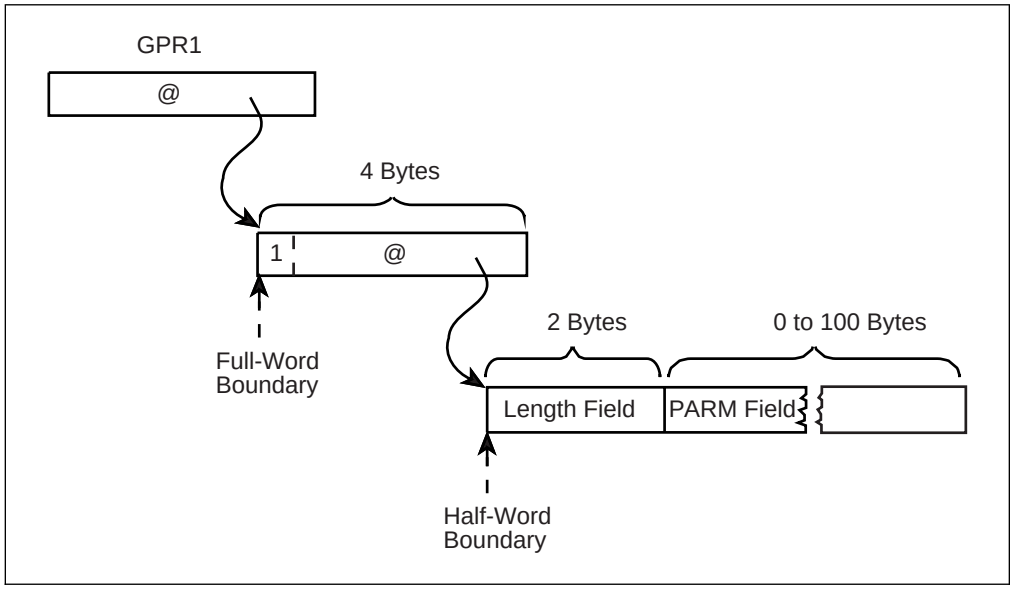


Figure 2-2. Primary Mode Parameter List

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 on page 2-11 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.

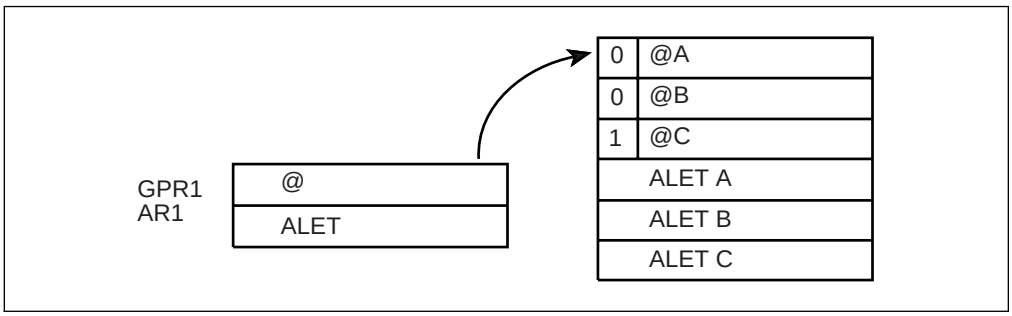


Figure 2-3. AR Mode Parameter List

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.

© Copyright IBM Corp. 1988, 1999

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 *OS/390 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.

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.

Stopping and Restarting a Subtask (STATUS Macro)

To stop a subtask means to change its dispatchability status from ready to not ready. You might need to stop a subtask that is currently ready or running, and then to restart, or make ready, that subtask. Stopping a subtask is a programming technique to control the dispatchability of other related tasks or subtasks in a multi-tasking environment when the tasks are in problem state.

To stop all subtasks of an originating task, issue the STATUS macro with the STOP parameter. To stop a specific subtask, issue the STATUS macro with the STOP,TCB parameter, which identifies a specific subtask.

To restart the stopped subtask or subtasks, issue STATUS START. As with STATUS STOP, use the TCB parameter to restart a specific subtask.

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.

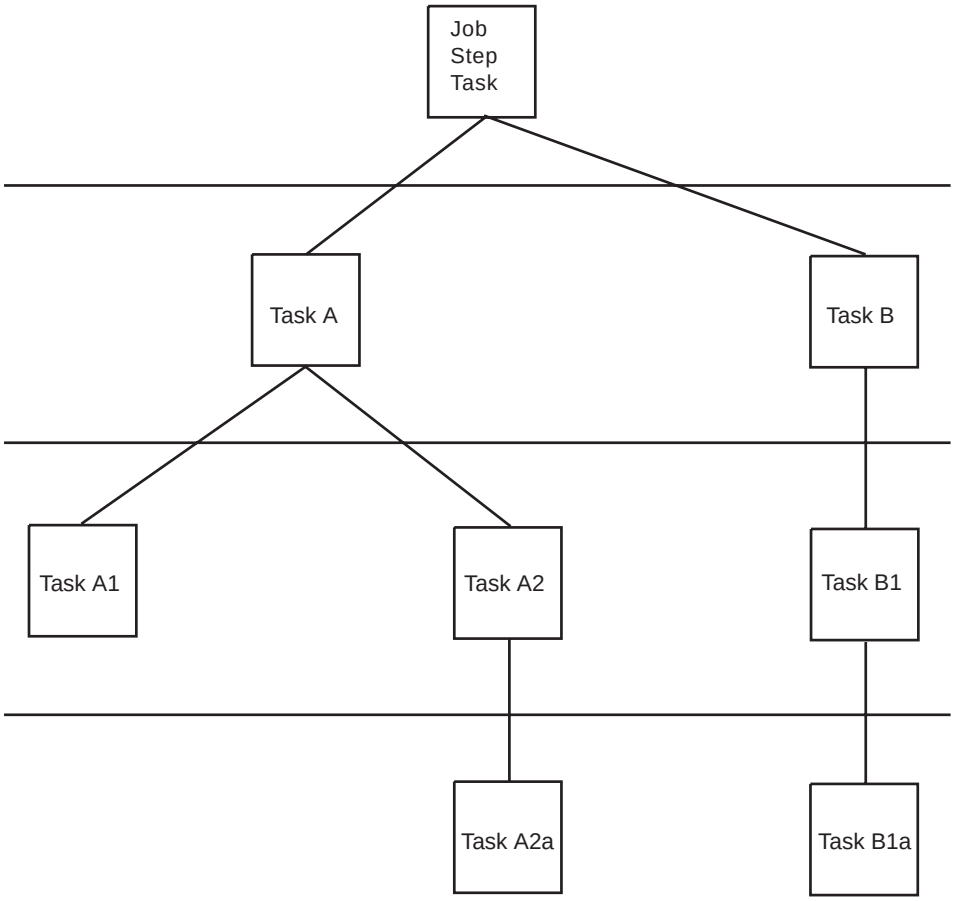


Figure 3-1. Levels of Tasks in a Job Step

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 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 terminated. 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. This chapter also includes a description of the macro used to request services from run-time library services (RTLS), a function that eases installation by allowing you to eliminate STEPLIBs from the JCL that runs your applications. In place of STEPLIBs, the new macro connects to and loads from a given RTLS logical library.

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

The assembler 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; *DFSMS/MVS Program Management* contains information about the linkage editor control cards.

© Copyright IBM Corp. 1988, 1999

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.

For more information on the linkage stack, see “System-Provided Linkage Stack” on page 2-3.

Floating Point Considerations

The application program and run-time environment are responsible for managing the contents of the Floating Point Control (FPC) register. The system will normally not change the FPC register settings of an existing MVS task or SRB.

The S/390 linkage convention for the Floating Point Registers and the FPC register in described in Chapter 2, “Linkage Conventions” on page 2-1. To summarize the convention, FPRs 0 to 7 are volatile and FPRs 8 to 15 are non-volatile across a call. The FPC register is non-volatile except for two fields: the IEEE exception flags and the DXC, which are volatile.

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

Passing Control Between Programs with All Registers Intact

The CSRL16J callable service allows you to transfer control to another program running under the same request block (RB) as the calling program. The CSRL16J callable service functions much like a branch instruction except that you can specify the contents of all 16 registers when you transfer control. You do not have to use one register to specify the address of the target routine, as you do with a branch instruction.

When you transfer control to the other routine, use the CSRL16J callable service to:

- Define the entry characteristics and register contents for the target routine.
- Optionally free dynamic storage associated with the calling program.

When the CSRL16J callable service is successful, control transfers to the target routine. After the target routine runs, it can transfer control to any program running under the same RB, including the calling program.

Defining the Entry Characteristics of the Target Routine

Before calling CSRL16J, you must build the L16J data area to form a parameter list that defines the entry characteristics and register contents for the target routine. Include the CSRYL16J mapping macro to map data area L16J. To build the L16J parameter list, first initialize the parameter list with zeroes and then fill in the desired fields. This processing ensures that all fields requiring zeroes are correct. You can specify the following characteristics for the target routine in the indicated fields of data area L16J:

L16JLENGTH Length of the L16J parameter list. Initialize this field with constant L16J_LEN.

L16JGRS General purpose registers (GPRs) 0-15 on entry to the target routine.

L16JARS Access registers (ARs) 0-15 on entry to the target routine, if you set the L16JPROCESSARS bit on.

L16JPSW Includes the following PSW information for the target routine. See *Principles of Operation* for more information about the contents of the PSW.

- PSW address and AMODE
- PSW ASC mode — primary or AR
- PSW program mask
- PSW condition code

APF-authorized callers, callers in supervisor state, PSW key 0-7, or PKM allowing key 0-7, can specify:

- PSW state - problem or supervisor
- PSW key.

For unauthorized callers, the PSW state and key of the calling program are used for the target routine.

L16JPROCESSARS A bit indicating whether or not you want to specify the contents of the access registers (ARs) for the target routine. Set the bit on if you want to specify the contents of the ARs. If you set the bit off, the access registers (ARs) contents are determined by the system.

When CSRL16J passes control to the target routine, the GPRs contain:

Register	Contents
0-15	Values specified by the caller

If the L16JPROCESSARS bit is set on, when CSRL16J passes control to the target routine the access registers (ARs) contain:

Register	Contents
0-15	Values specified by the caller

If the L16JPROCESSARS bit is set off, when CSRL16J passes control to the target routine the access registers (ARs) contain:

Register	Contents
0-1	Do not contain any information for use by the routine
2-13	The contents are the same as they were when the caller issued the CSRL16J callable service.
14-15	Do not contain any information for use by the routine

Freeing Dynamic Storage Associated with the Caller

If the calling program has a dynamic storage area associated with it, you can specify that CSRL16J free some or all of this storage area before it transfers control to the target routine. In the L16J parameter list, specify the following fields:

L16JSUBPOOL	Specify the subpool of the area that you want the system to free.
L16JLENGTHTOFREE	Specify the length, in bytes, of the dynamic storage area you want the system to free.
L16JAREATOFREE	Specify the address of the dynamic storage area you want the system to free. Make sure that the address is on a doubleword boundary. Otherwise, the service ends with an abend code X'978'. See *OS/390 MVS System Codes* for information on abend code X'978'.

The system frees the storage only when the CSRL16J callable service is successful.

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 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.

Figure 4-3. Characteristics of Load Modules

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

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.

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 *DFSMS/MVS Program Management* 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 4-12, 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'0FFF'). 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. Each load module or program object resides in a library, either a partitioned data set (PDS) or partitioned data set extended (PDSE). A load module resides in a PDS, and a program object resides in a PDSE. In most cases, references in this book to load modules apply to both load modules and program objects. Any exceptions are specifically noted. As the control program brings the load module into virtual storage, it 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 macros.

Location of the Load Module

Load modules and program objects can reside in 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 *OS/390 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 allowing key 0 - 7) invokes an unauthorized program, the unauthorized program must reside in an APF-authorized library. See

“APF-authorized Programs and Libraries” on page 4-29 for more information about APF-authorized libraries.

- 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 *DFSMS/MVS Using 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. 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,...
```

Use extreme caution when specifying duplicate module names. Even if you code the DCB parameter, the wrong module can still receive control. For example, suppose there are two modules with the same name you want to invoke, one after the other. To distinguish between them in this example they are called PROG2 and PROG2'. PROG1 issues a LOAD for PROG2 and BALRs to it. PROG2 issues a LINK specifying a DCB for the library with the other copy of PROG2 (which we are calling PROG2'). The LINK will find a useable copy of PROG2 in the Job Pack Area and invoke it again, regardless of the DCB parameter. PROG2 again issues a LINK for PROG2'. This time the copy of PROG2 in the Job Pack Area is marked "not reusable" and PROG2' is loaded using the DCB parameter and given control.

The problem encountered in the example above could be avoided by any one of the following sequences:

- PROG1 links to PROG2 and PROG2 links to PROG2'
- PROG1 loads and branches to PROG2. PROG2 loads and branches to PROG2'
- PROG1 links to PROG2 and PROG2 loads and branches to PROG2'

Once a module has been loaded from a task library, the module name is known to all tasks in the address space and may be used as long as the module is considered usable. Generally speaking, reenterable modules are always usable. Serially reusable modules are usable when they are currently in use.
Non-reentrant, non-serially reusable modules are considered usable for LOAD if the use count is zero. A module is considered usable for ATTACH, LINK, or XCTL if it has not been marked NOT REUSABLE by a previous ATTACH, LINK, or XCTL. The use count is not considered.

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. See *DFSMS/MVS Macro Instructions for Data Sets* for more information about the BLDL macro.

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. 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/E 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 use 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 non-reusable 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.

Note: A program that is LINKed to will get control with the caller's Floating Point Registers and Floating Point Control register. The S/390 linkage convention applies. For more information, see Chapter 2, Linkage Conventions.

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

When a module is reenterable, 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 non-reusable, 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.

Passing Control without Return

Use the XCTL or XCTLX macro to pass control to a target load module when return of control is not required. You can also pass control using a branch instruction. However, when you pass control in this manner, you must ensure that multiple uses of non-reusable or serially reusable modules **does not occur**. 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 non-reusable 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 will not be returned, the XCTL issuer **must** load the address of the old save area into register 13 prior to issuing the XCTL. 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.

Note: A program that is XCTLed to will get control with the caller's Floating Point Registers and Floating Point Control register. The program that issued the XCTL macro is not returned to, instead the XCTLed program will return to the program that caused the issuer of the XCTL macro to run. The S/390 linkage convention applies except that the non-volatile FPRs and FPC register that must be restored are different. The issuer of the XCTL macro must restore its caller's non-volatile FPRs and FPC register before issuing the XCTL (just as if it were returning to its caller). For more information on linkage conventions, please refer to Chapter 2, Linkage Conventions.

When using the XCTL or XCTLX macro, pass parameters to the target module in a parameter list. In this case, however, the parameter list (or the parameter data) must be established in remote storage, 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 the control program using a control parameter list. The control program loads a copy of the target 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 use count for the copy of the target load module and subtracts one from the use 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 use count will be lowered for the wrong load module copy. And remember, when the use 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 executes, independently of how control was passed, and issues an XCTL or XCTLX macro

when it is finished (step 2) to pass control to target load module C. The control program knowing only of load module A, lowers the use 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.

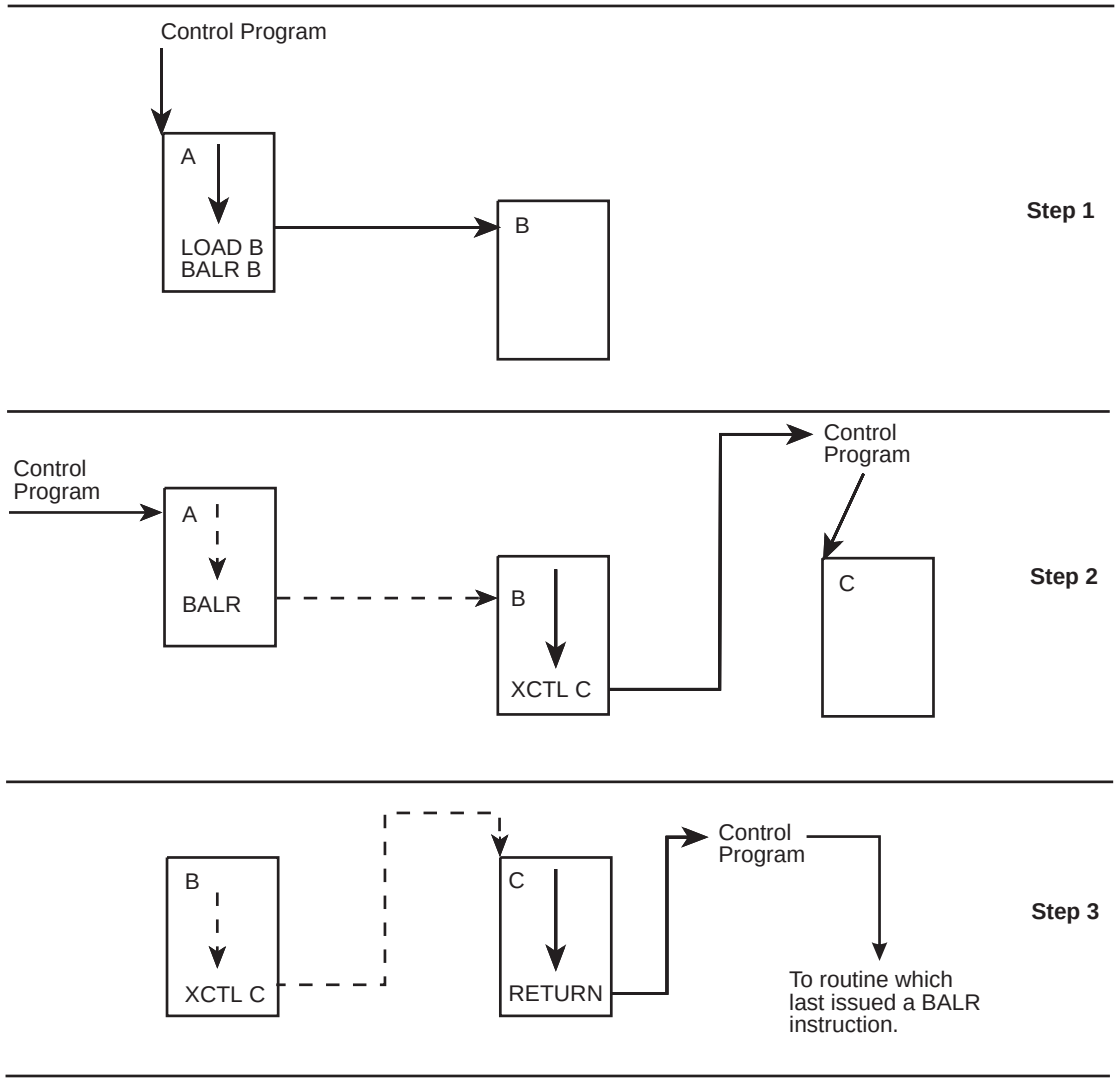


Figure 4-20. Misusing Control Program Facilities Causes Unpredictable Results

Two methods are available for ensuring that the proper use 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.

Note: The control program abnormally terminates the task if the XCTL issuer added entries to the linkage stack and did not remove them before issuing the XCTL.

APF-authorized Programs and Libraries

The authorized program facility (APF) helps your installation protect the system. APF-authorized programs can access system functions that can affect the security and integrity of the system. APF-authorized programs must reside in APF-authorized libraries, which are defined in an APF list.

Unauthorized programs can issue the CSVAPF macro to:

- Determine whether or not a library is in the APF list
- Determine the current format (dynamic or static) of the APF list
- Obtain a list of all library entries in the APF list.

APF also prevents authorized programs (supervisor state, APF-authorized, PSW key 0-7, or PKM 0-7) from accessing a load module that is not in an APF-authorized library. The application development books for programmers who use authorized programs provide more information about APF authorization.

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. The IDENTIFY macro cannot be issued by supervisor call routines, SRBs, 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. You are responsible for not duplicating a member name or an alias in any of the libraries. Duplicate names cause the system to return a return code of 8.

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

If an authorized program creates an alias for a module in the pageable link pack area or active link pack area, the IDENTIFY service places an entry for the alias on the active link pack area queue. If an unauthorized user creates an alias for a

module in the pageable link pack area or active link pack area, the IDENTIFY service places an entry for the alias on the job pack queue of the requesting job.

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.

Retrieving Information About Loaded Modules

Both the CSVINFO and CSVQUERY macros return information about loaded modules. A **loaded module** is a load module that has been loaded into storage. Use CSVQUERY if you need information about a particular loaded module or if your program is running in access register (AR) mode. Use CSVINFO to obtain information about a group of loaded modules or when you want information about the loaded module associated with a particular program request block (PRB) or information that the CSVQUERY macro does not provide.

The following information is available only through the CSVINFO macro:

- Whether the entry point is an alias created using the IDENTIFY macro.
- The starting address of every extent and the number of extents for loaded modules with multiple extents. (CSVQUERY provides only the entry point address and the total module length.)
- The load count, system count, and total use count for the loaded module.
- The name of the major entry point, if the entry point is an alias.
- The full entry point name for modules with names longer than 8 characters.

In addition to the information you request, the CSVINFO macro returns the file name for modules in the OpenMVS file system.

Using the CSVINFO Macro

The CSVINFO macro provides information about loaded modules associated with a job step or a task. You can invoke CSVINFO from a program or an IPCS exit.

Note: IBM recommends that you use the CSVINFO macro rather than write your own program to scan control blocks for information about loaded modules. Using the CSVINFO macro enables you to retrieve module information without depending on the details or structures of data areas.

The CSVINFO service requires a user-written module information processing routine (MIPR). The CSVINFO service obtains information about loaded modules and uses the CSVMODI data area to pass that information to the MIPR. The MIPR examines this data and returns control to CSVINFO, either requesting information about an additional loaded module or indicating that no more information is needed.

This loop continues until the CSVINFO service has returned to the MIPR all requested information or all available information.

For example, if you request information about all loaded modules in your job pack area (JPA), the CSVINFO service uses the CSVMODI data area to pass information about the first loaded module to the MIPR. The MIPR processes the information and returns control to CSVINFO to obtain information about the next loaded module in the JPA. Processing continues until CSVINFO indicates that all information has been obtained or until the MIPR determines that no more information is required.

When you issue the CSVINFO macro, use the FUNC parameter to specify the information you want, and the ENV parameter to specify whether CSVINFO is being issued from a program or from an IPCS exit. Use the MIPR parameter to pass the address of your MIPR. You can pass 16 bytes of information to the MIPR using the USERDATA parameter. Information could include register contents, parameter list addresses, or other information your MIPR requires. CSVINFO places your user data into the CSVMODI data area.

References

The CSVMODI data area serves as the interface between the CSVINFO service and the MIPR. For more information about the CSVMODI mapping macro, see *OS/390 MVS Data Areas, Vol 1 (ABEP-DALT)*.

OS/390 MVS IPCS Commands explains how to verify the correct use of the CSVINFO macro in an IPCS exit. See the TRAPON, TRAPOFF, and TRAPLIST subcommand descriptions.

OS/390 MVS IPCS Customization provides information about writing IPCS exits.

Figure 4-21 on page 4-32 shows the processing that occurs when your program or exit issues the CSVINFO macro. The numbered steps are explained below:

1. The application or IPCS exit invokes the CSVINFO macro.
2. CSVINFO retrieves the module information you want.
3. CSVINFO places the information into the CSVMODI data area.
4. CSVINFO passes control to your MIPR.
5. The MIPR reads the information that is in the CSVMODI data area.
6. The MIPR places the information into your storage or otherwise processes the information.
7. The MIPR sets a return code for CSVINFO:
   - A return code of zero to request information about another loaded module
   - A nonzero return code to indicate that no more information is needed.
8. The MIPR returns control to CSVINFO.
9. Steps 2 through 8 are repeated until the MIPR indicates to CSVINFO that no more information is needed, or CSVINFO indicates to the MIPR that all information has been retrieved.

10. CSVINFO sets a return code and returns control to your program when the MIPR passes CSVINFO a return code indicating that no more information is needed, or when CSVINFO has passed all the information to the MIPR.
11. The application or IPCS exit continues processing.

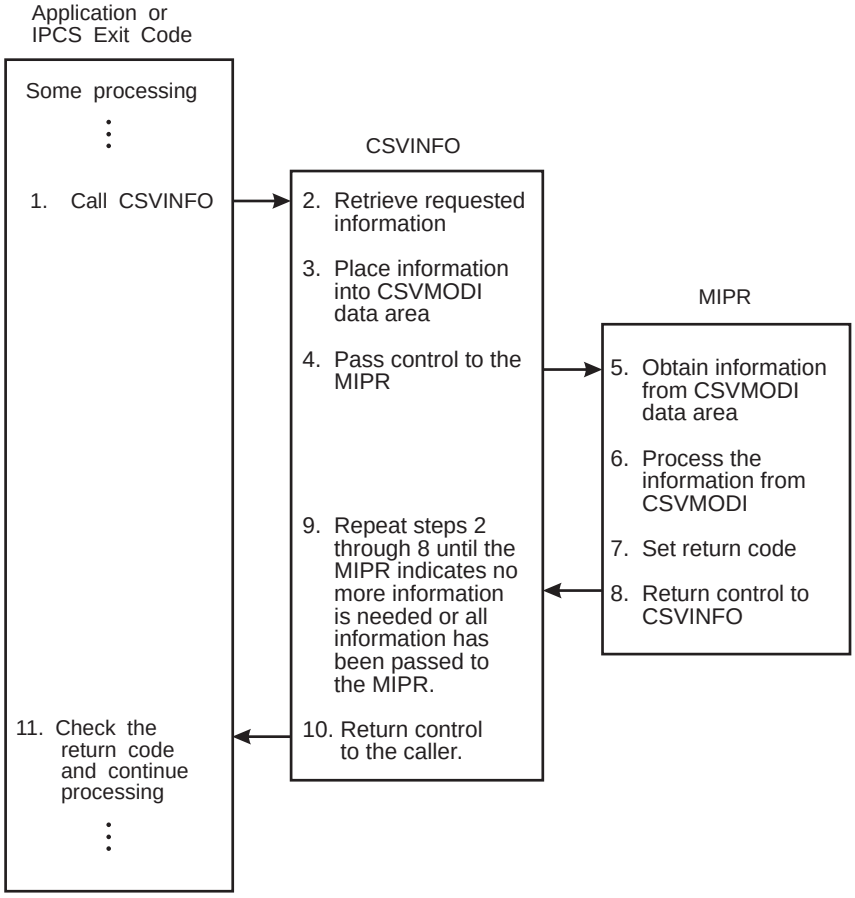


Figure 4-21. Processing Flow for the CSVINFO Macro and the Caller's MIPR

Serialization

Information about loaded modules in common storage is serialized by the LOCAL and CMS locks. Information about other loaded modules is serialized by the LOCAL lock. When the CSVINFO service runs with serialization, you are guaranteed that the information CSVINFO obtains is not in the process of being updated.

If your program runs in problem state and invokes the CSVINFO macro, your program cannot hold the appropriate locks and the CSVINFO service does not obtain them. Thus, the CSVINFO service retrieves information without serializing on it. If you are requesting information about loaded modules in common storage or if multi-tasking is taking place in your address space, the module information you request might be changing while the CSVINFO service is retrieving information. In rare instances, the CSVINFO service could return incorrect information or end abnormally.

If your program runs in supervisor state and invokes the CSVINFO macro, the CSVINFO service obtains the appropriate locks if your program does not already hold them.

Coding a MIPR for the CSVINFO Macro

This section contains information about coding a MIPR.

Installing the MIPR

You can either link-edit your MIPR with the program that invokes the CSVINFO macro or include the MIPR in mainline code.

MIPR Environment

The MIPR receives control running under the unit of work that invoked the CSVINFO macro, in the following environment:

Minimum authorization: Problem state and any PSW key.
Dispatchable unit mode: Task or SRB
Cross memory mode: PASN=SASN=HASN
AMODE: 31-bit
ASC mode: Primary
Interrupt status: Enabled for I/O and external interrupts
Control parameters: Control parameters must be in the primary address space.

Recovery for MIPR Provided by CSVINFO

Figure 4-22 shows the recovery environment that the CSVINFO service establishes for itself and the MIPR.

Figure 4-22. CSVINFO Recovery

ENV Keyword	Caller State/Key	Recovery Provided
ENV=MVS	Supervisor state	The caller's MIPR gets control with CSVINFO's FRR in effect.
	Problem state	The caller's MIPR gets control with CSVINFO's ESTAE routine in effect.
ENV=IPCS	Problem or supervisor state	No recovery provided

Entry Specifications

The MIPR gets control through standard branch entry linkage. Input to the MIPR is the address of the CSVMODI data area, containing information from the CSVINFO service.

Registers at Entry

When the MIPR receives control, the general purpose registers (GPRs) contain the following information:

GPR	Contents
0	Does not contain any information for use by the routine
1	Address of the CSVMODI data area
2 - 12	Does not contain any information for use by the routine
13	Address of a standard 72-byte save area
14	Return address to the CSVINFO service
15	Entry point address of MIPR

Return Specifications

Upon return from MIPR processing, you must ensure that the register contents are as follows:

Registers at Exit

GPR	Contents
0-1	The MIPR does not have to place any information into these registers, and does not have to restore their contents to what they were when the MIPR received control.
2-13	The MIPR must restore the register contents to what they were when the MIPR received control.
14	Return address to the CSVINFO service
15	Return code from the MIPR

Note: The CSVINFO service continues processing until either of the following occurs:

- It receives a non-zero return code from the MIPR.
- It has returned all available data

When CSVINFO receives a non-zero return code, it returns control to the program that invoked the CSVINFO macro.

CSVINFO Service Coding Example

The CSVSMIPR member of SAMPLIB contains a coded example of the use of the CSVINFO service and its associated MIPR. The sample program is a reentrant program that has its MIPR included within the same module.

Using CSVRTLS to Request Run-Time Library Services (RTLS)

Run-time library services (RTLS) enables you to eliminate STEPLIBs from the JCL that runs your applications. By eliminating STEPLIBs, you reduce the installation management your applications require, as well as the system overhead involved in searching STEPLIB data sets when loading modules into storage. In place of STEPLIBs, the CSVRTLS macro connects to and loads from a given RTLS logical library.

Both authorized and unauthorized programs can use CSVRTLS, but only authorized programs can use the TCBADDR parameter or connect to more than 32 libraries per address space. See the minimum authorization requirements listed in *OS/390 MVS Programming: Assembler Services Reference* for options available to authorized programs.

Language Environment applications currently can exploit RTLS by using the Language Environment run-time options RTLS(ON), LIBRARY(le_run-time_lib), and VERSION(version), which identify the RTLS logical library to be connected. During application startup, the Language Environment element connects to the specified library using the macro CSVRTLS REQUEST=CONNECT, loads from the specified library using CSVRTLS REQUEST=LOAD, and disconnects from the logical library using CSVRTLS REQUEST=DISCONNECT. Language Environment runs as an extension to the application, allowing it to load from any library the application is authorized to access.

An operator command, SET RTLS=xx, allows dynamic configuration of the RTLS library definitions. Each of these library definitions, or logical libraries, identifies a load library search order that is to be used when an application connects to and loads from a specific logical library. Each logical library search order definition contains one or more physical library definitions that can consist of one or more load library data sets. Caching to common storage is performed on a physical library basis to provide optimal performance for commonly used libraries. Each physical library is either authorized or unauthorized. This determination is made when the library is added or replaced; it is not changed or affected by changes to the APF status of individual data sets.

The SET RTLS=xx command, like the RTLS system parameter at IPL, uses the CSVRTLxx parmlib members. The SET command allows for the introduction of new physical and logical libraries, removal of physical and logical libraries, or the replacement of libraries. The DISPLAY RTLS command displays the current status of the RTLS environment. The information supplied through DISPLAY RTLS optionally includes the physical and logical libraries in use, the users of the logical libraries, and the cache utilization for a given library or for all libraries.

The contents supervision services XCTL, LINK, ATTACH, IDENTIFY, CSVQUERY, and CSVINFO all work with modules loaded through the CSVRTLS REQUEST=LOAD service.

For more information about the CSVRTLS macro, see *OS/390 MVS Programming: Assembler Services Reference*.

Chapter 5. Understanding 31-Bit Addressing

Note

This chapter documents the programming considerations for running 31-bit addressing mode programs on previous versions and releases of MVS. Because this information might be useful for programmers who maintain or update legacy programs, the chapter is preserved to reflect the programming environment of previous MVS versions. If you intend to design and code a new program to run on OS/390 MVS releases, consider the following:

- Always design a program to run in 31-bit addressing mode, to take full advantage of the virtual storage capacity of MVS.
- Use the IBM High Level Assembler, instead of Assembler H, to assemble the new program. As of MVS/SP 5.2, Assembler H is not supported.
- Use the DFSMS/MVS program management binder, instead of the MVS/DFP linkage editor and loader, to prepare the program for execution.

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 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.

© Copyright IBM Corp. 1988, 1999

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.

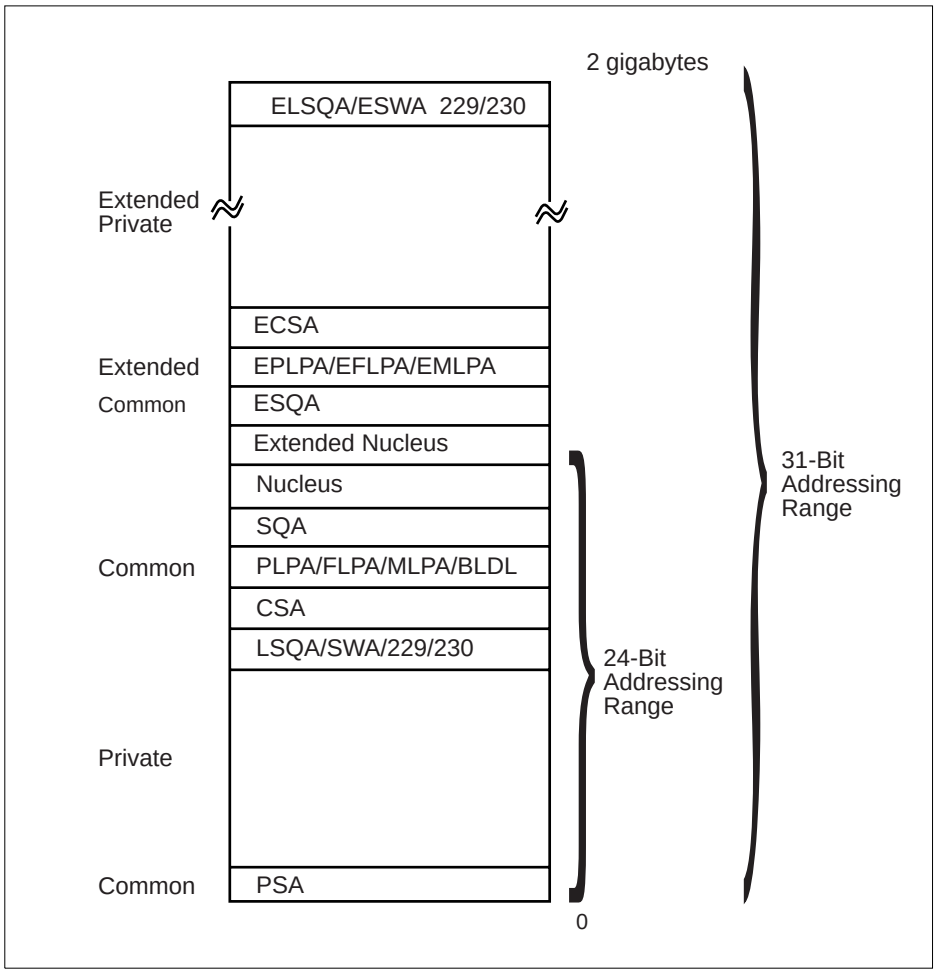


Figure 5-1. Two Gigabyte Virtual Storage Map

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 or an assembler that provides equivalent function, 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

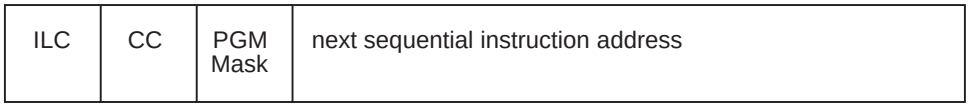
The processor is sensitive to the addressing mode that is in effect (the setting of the PSW A-mode bit). The current PSW controls instruction sequencing. The instruction address field in the current PSW contains either a 24-bit address or a 31-bit address depending on the current setting of the PSW A-mode 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.

First operand register (24-bit addressing mode)



ILC - instruction length code
CC - condition code
PGM Mask - program mask

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.

First operand register (31-bit addressing mode)

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 5-23.

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.

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.)

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 Storage Management (DFSMSdfp).

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 helps you identify programs that need to be changed. You will need to consult one or more versions of the MVS conversion notebook. The versions you need depend on your current version of MVS and the version of MVS to which you are migrating. See the appropriate version of the MVS conversion notebook for your migration.

Planning for 31-Bit Addressing helps you determine what changes to make to a module you are converting to 31-bit addressing and indicates 31-bit address considerations 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.

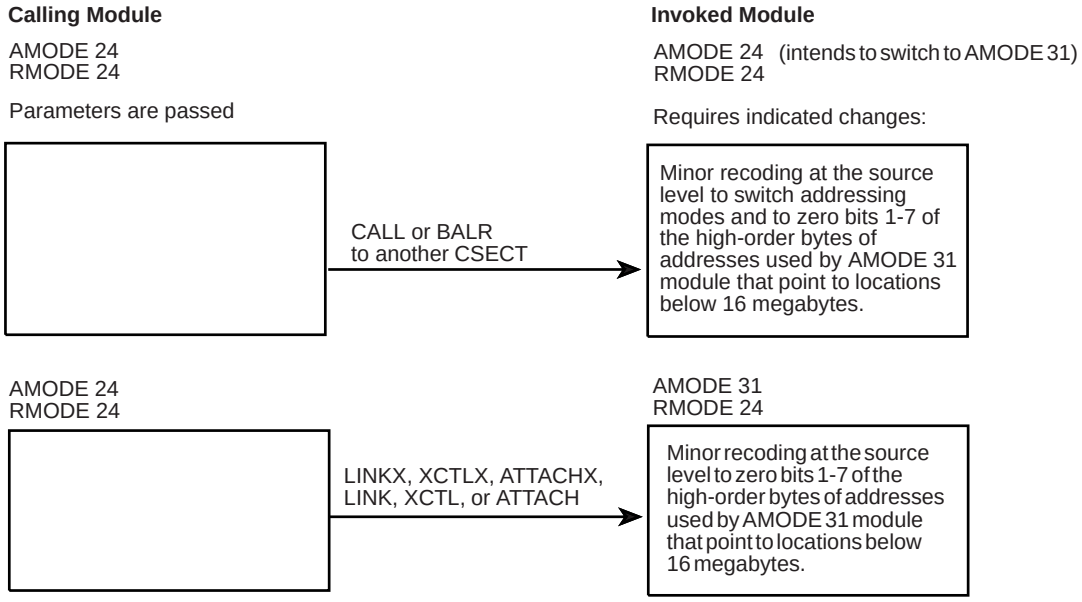


Figure 5-2. Maintaining Correct Interfaces to Modules that Change to AMODE 31

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.)

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

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 5-23). 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 5-23) and use BASSM. (The moved module will use BSM to return.)	Create a linkage assist routine (described in "Establishing Linkage" on page 5-23).
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 5-23).

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 5-23.

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 or the VRU, VRC, RU, and RC forms of GETMAIN and FREEMAIN, or the corresponding functions on STORAGE. In addition, you can obtain storage above 16 megabytes by using CPOOL services. 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 SDWA for PSW-related information, they should refer to SDWAEC1 (EC mode PSW at the time of error) and SDWAEC2 (EC mode PSW of the program request block (PRB) that established the ESTAE-type recovery.

  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 5-23 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 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 or MVS/ESA expansions.

The SPLEVEL macro allows programmers to change the value of the SET symbol. The SPLEVEL macro shipped with OS/390 MVS sets a default value of 5 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, or SPLEVEL=3, 4, or 5 to obtain MVS/ESA 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     *
```

OS/390 MVS Programming: Assembler Services Guide and *OS/390 MVS Programming: Assembler Services 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 *OS/390 MVS Diagnosis: Tools and 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 only combination of AMODE and RMODE that is not valid is AMODE 24/ RMODE ANY.

The following errors will keep your program from assembling:

- 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.

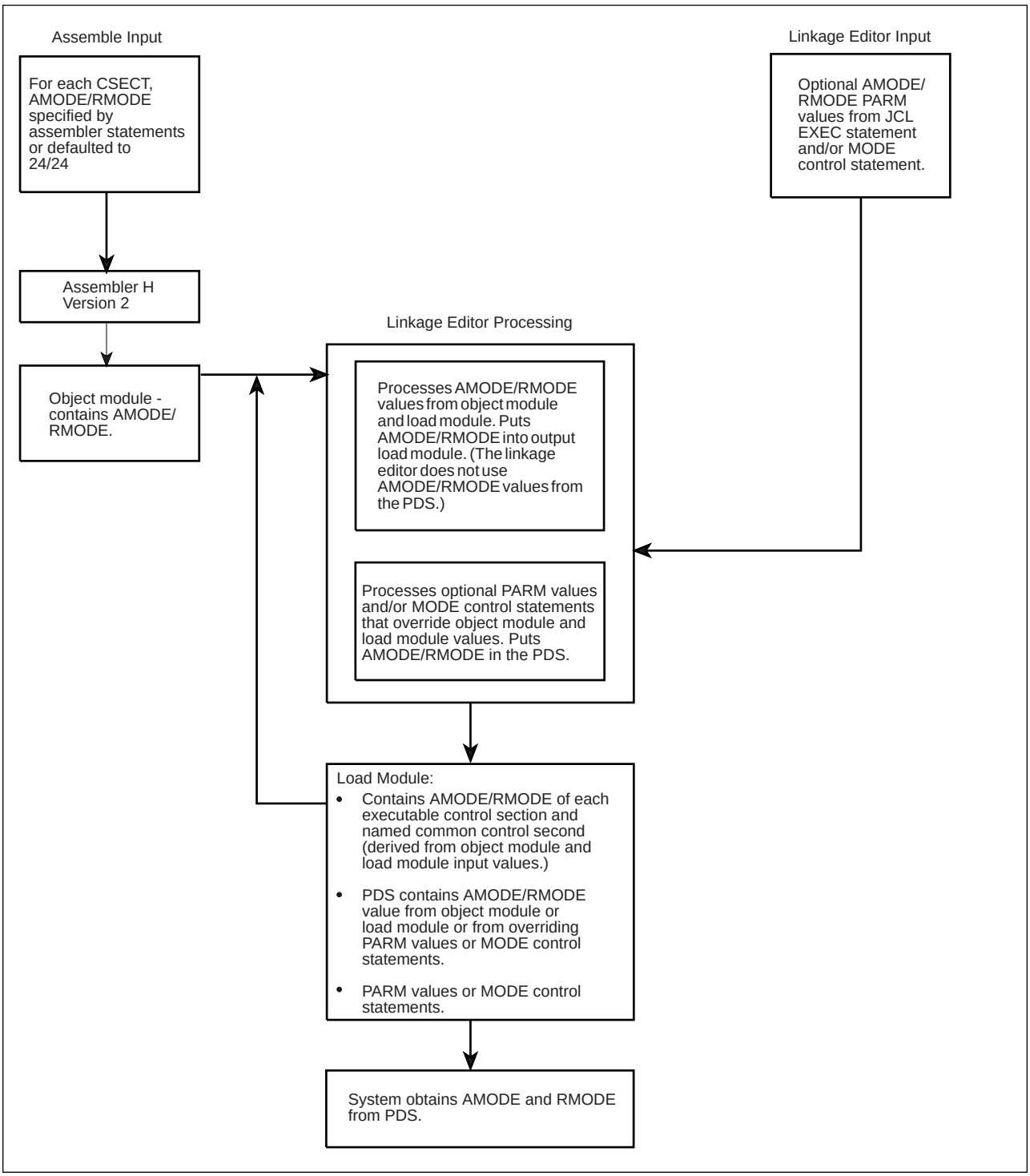


Figure 5-5. AMODE and RMODE Processing by the Linkage Editor

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.

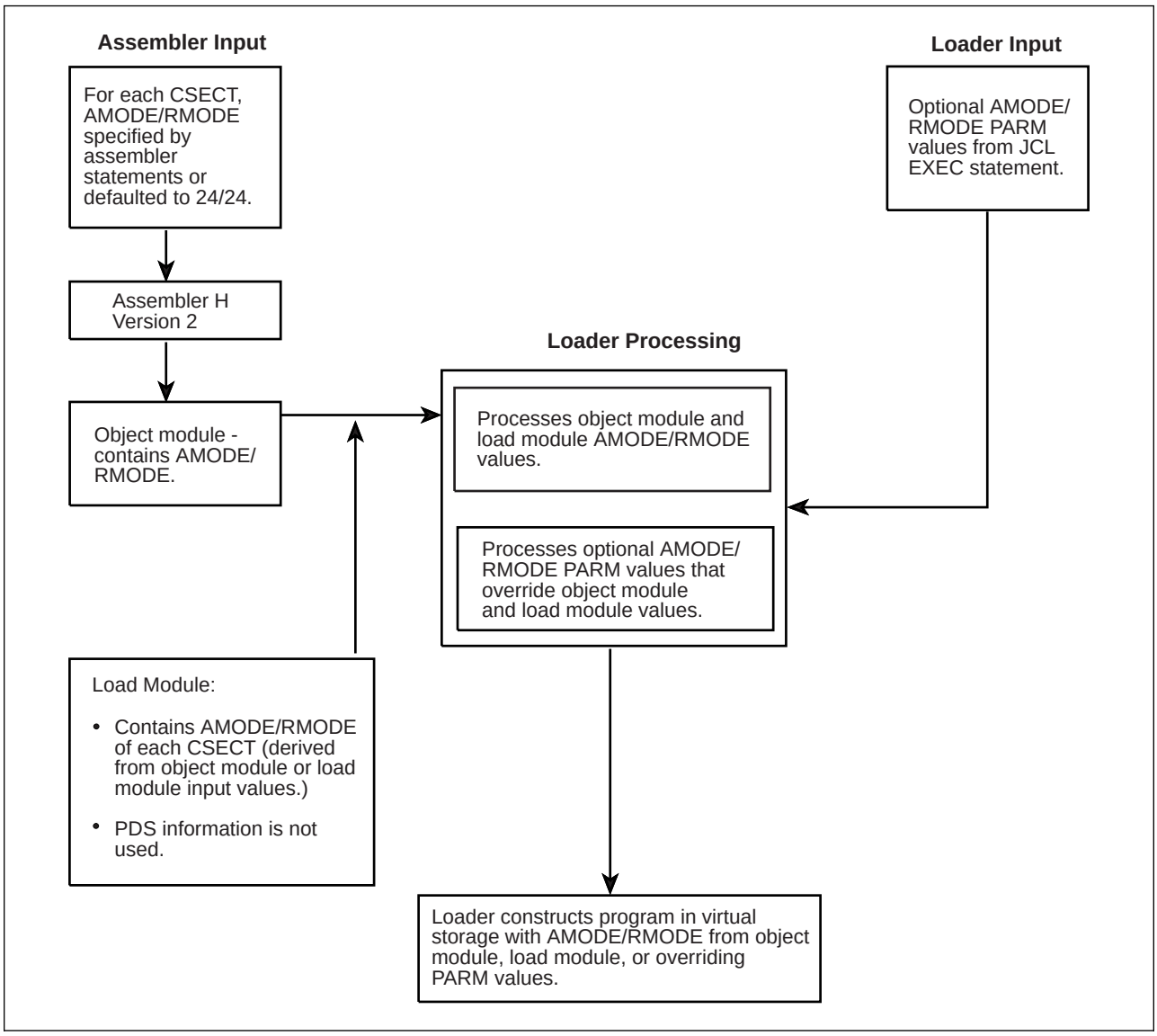


Figure 5-6. AMODE and RMODE Processing by the Loader

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 SVCs, 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 *DFSMS/MVS Using 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.
- The CIRB macro and the stage 2 exit effector. The CIRB macro is described in *OS/390 MVS Programming: Authorized Assembler Services Guide* and *OS/390 MVS Programming: Authorized Assembler Services Reference ALE-DYN*.
- 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

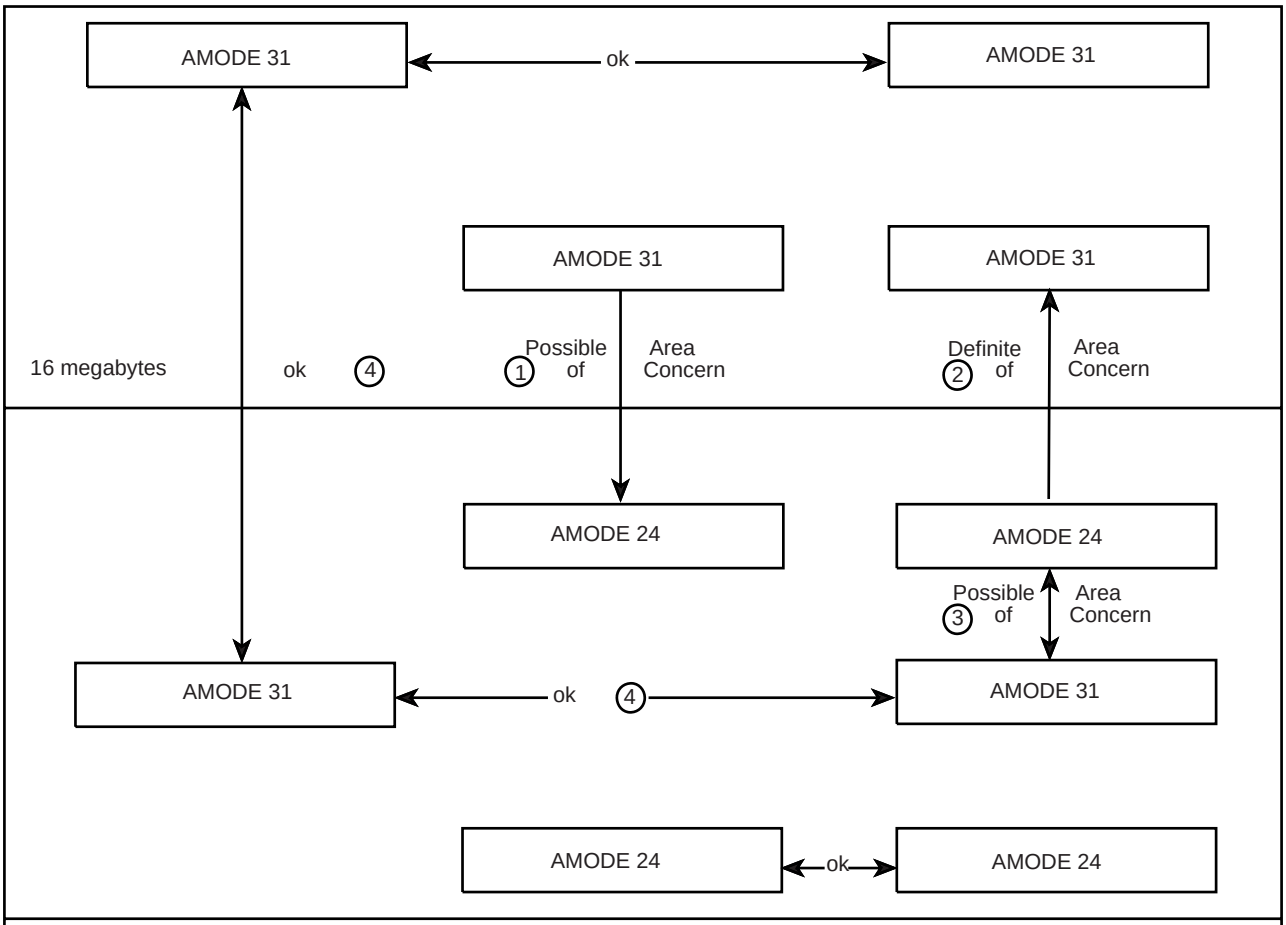
This section describes the mechanics of correct linkage in 31-bit addressing mode. Keep in mind that there are considerations other than linkage, such as locations of areas that both the calling module and the invoked module need to address.

Linkage in MVS systems that use 31-bit addressing (MVS/XA, MVS/ESA and OS/390) 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.

AMODE 31



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

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

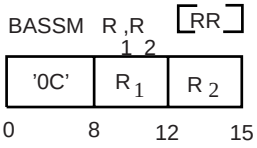
- Use the branching instructions (BASSM and BSM)
- Use pointer-defined linkage

- Use supervisor-assisted linkage (ATTACH, ATTACHX, LINK, LINKX, XCTL, and XCTLX)
- Use linkage assist routines
- Use “capping.”

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.)



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.

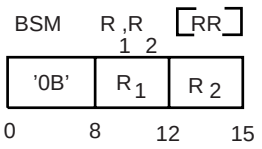
When the contents of the general register designated by the R_2 field are used, bit 0 of the register specifies the new addressing mode and replaces bit 32 of the current PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The new value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



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.

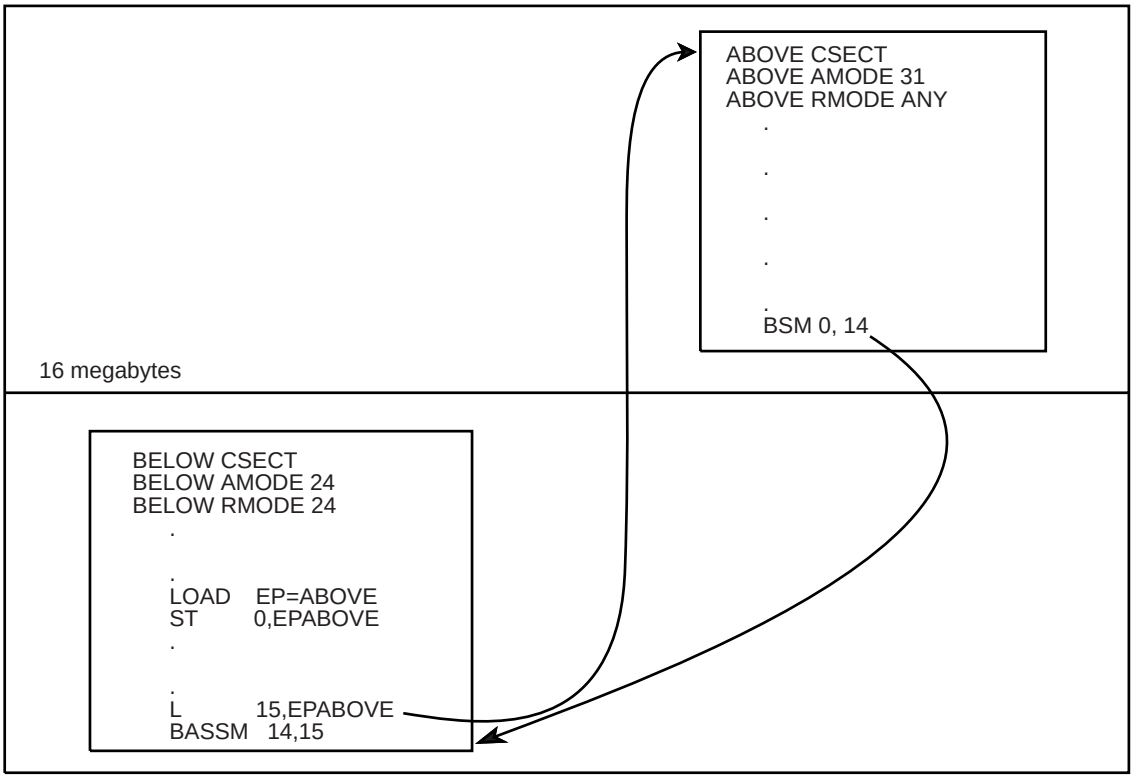


Figure 5-11. Using BASSM and BSM

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 in addition to 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.

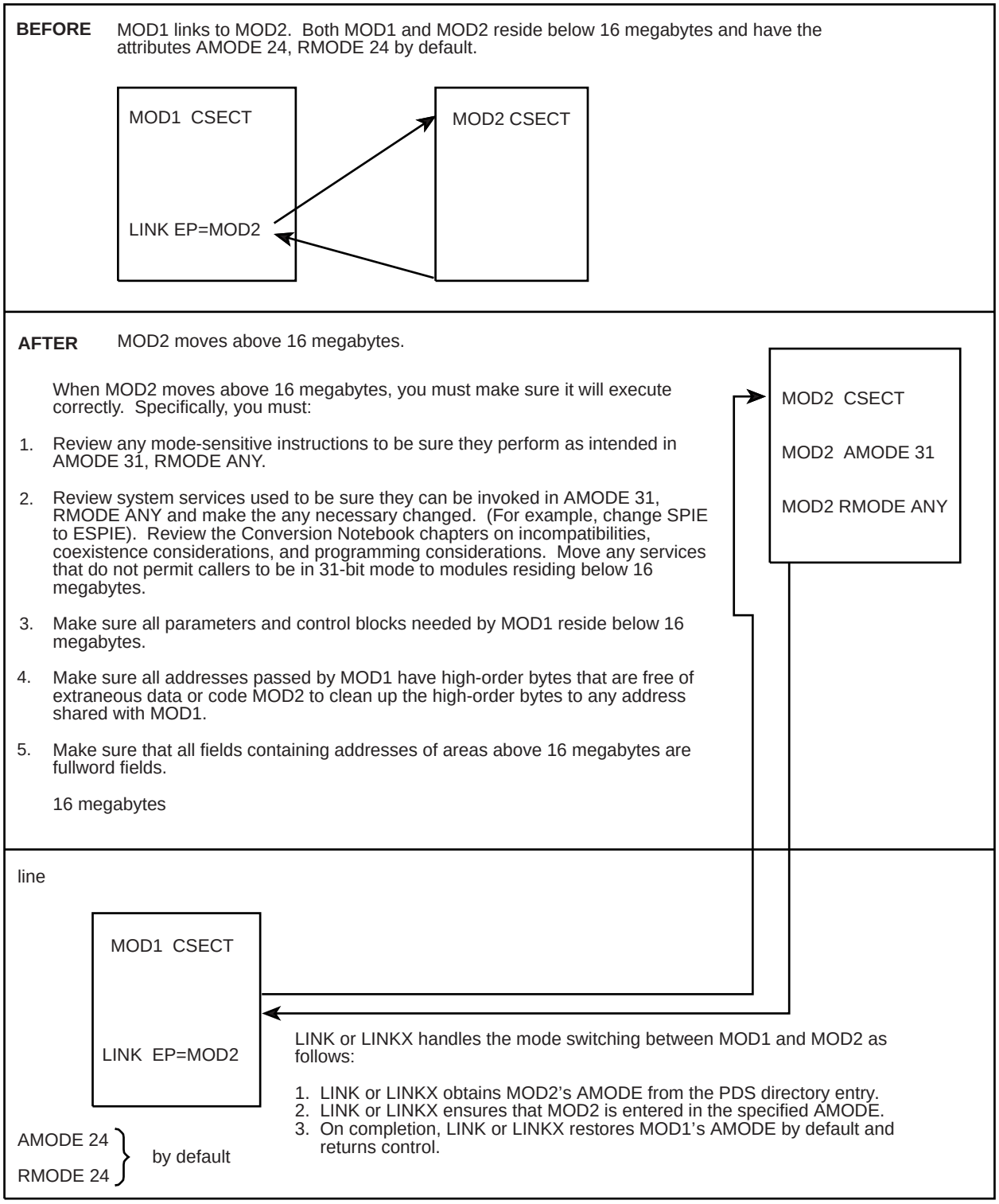


Figure 5-13. Example of Supervisor-Assisted Linkage

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

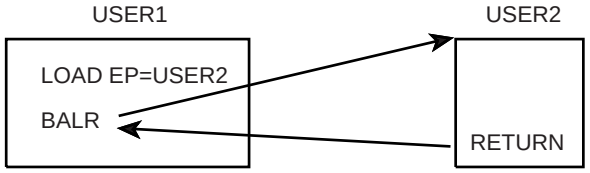
Figure 5-14 shows a “before” and “after” situation involving modules USER1 and USER2. USER1 invokes USER2 by using a LOAD and BALR sequence. The “before” part of the figure shows USER1 and USER2 residing below 16 megabytes and lists the changes necessary if USER2 moves above 16 megabytes. USER1 does 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. • Write a linkage assist routine called USER2. • Change USER3 (formerly USER2) as follows: - Make sure all control blocks and parameters needed by USER1 and USER2 are located below 16 megabytes line. - Check mode-sensitive instructions to be sure they perform the intended function in AMODE 31, REMODE 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. - Make sure that all fields containing addresses of areas above 16 megabytes are fullword fields.	• USER1 does not have to change the LOAD USER2 macro. • USER1 remains unchanged; new USER2 switches AMODEs and branches to USER3 (the former USER). - USER1 and USER2 are AMODE 24; they cannot access parameters or data above 16 megabytes. - USER3 was moved above 16 megabytes and has the attributes AMODE 31, REMODE ANY. - USER3 has the attributes AMODE 31, RMODE ANY. SPIE and some other system services will not work in AMODE 31.

AFTER

Changed Application

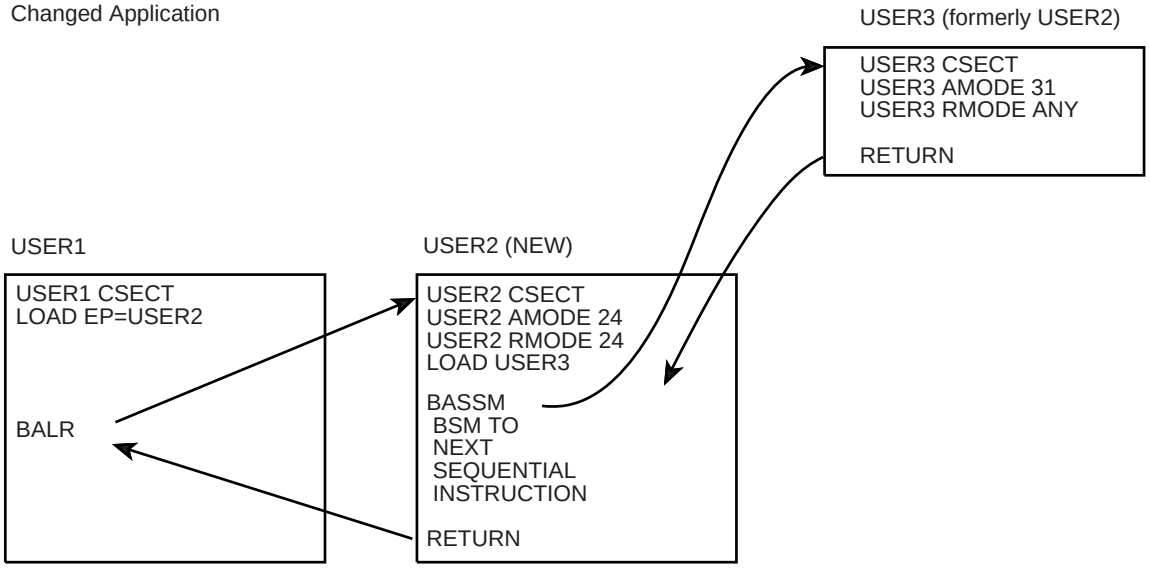


Figure 5-14 (Part 1 of 5). Example of a Linkage Assist Routine

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
*  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
```

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

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 3 of 5). Example of a Linkage Assist Routine

USER2 (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 4 of 5). Example of a Linkage Assist Routine

USER3 (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 5 of 5). 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. (*DFSMS/MVS Using 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” on page 5-39 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 5-23 for more information about this method.

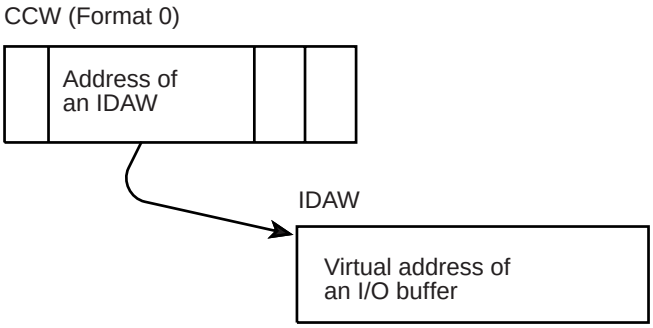
- Using the method shown in Figure 5-16 on page 5-41.

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.



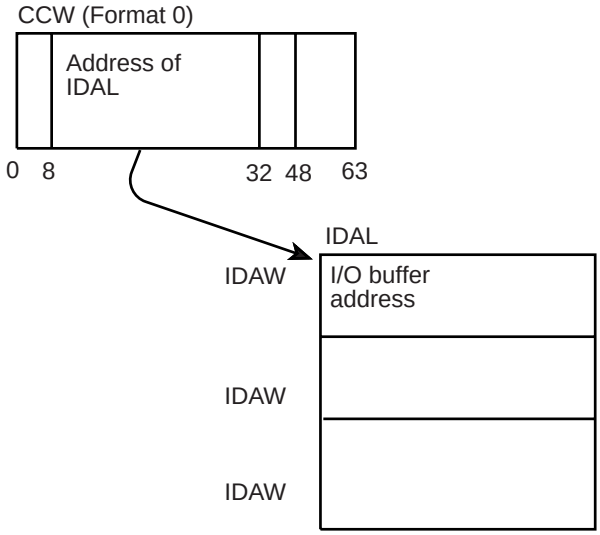
Any CCW that causes data to be transferred can point to a virtual IDAW. Virtual IDAW support is limited to non-VIO data sets.

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.

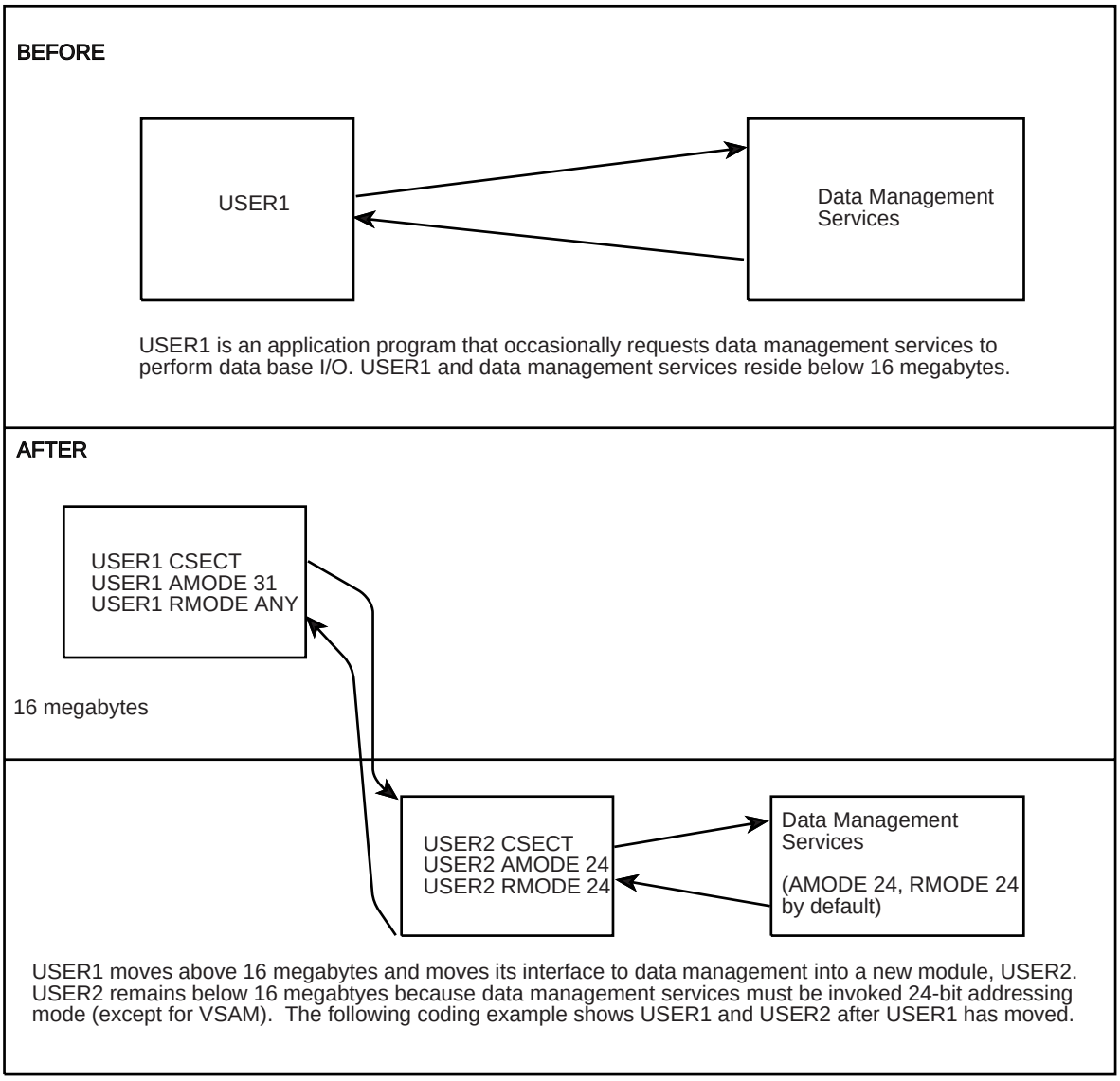


Figure 5-16 (Part 1 of 9). Performing I/O While Residing Above 16 Megabytes

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.

```
         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 2 of 9). 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.

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 3 of 9). 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.

```
         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
*                                      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 4 of 9). 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
```

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 5 of 9). 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 6 of 9). 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 7 of 9). 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 8 of 9). 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 9 of 9). 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 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.

Note: There is exception to the meaning of LOC=RES and LOC=(RES,ANY). A caller residing below 16 megabytes but running with 31-bit addressing can specify LOC=RES (either explicitly or by taking the default) or LOC=(RES,ANY) to obtain storage from a subpool supported only above 16 megabytes. In this case, the caller's AMODE determines the location of the virtual storage.

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, MVS/ESA and OS/390 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, MVS/ESA and OS/390 save 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, MVS/ESA and OS/390 nucleus is not V=R, their 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 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 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).

See *OS/390 MVS Programming: Authorized Assembler Services Guide* and *OS/390 MVS Programming: Authorized Assembler Services Reference ALE-DYN* 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:

- The WAIT, POST, and EVENTS macros introduce a strategic delay in the running of a program. This delay forces a program to wait using an event control block (ECB), 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.
- Pause, Release, and Transfer are services you can call, using a pause element (PE), to synchronize task processing with minimal overhead.

Figure 6-1 compares the use of the Pause, Release, and Transfer services with the WAIT and POST macros.

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

- 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.

  Under certain circumstances, you might want to use the RESERVE macro instead of ENQ to serialize use of resources.
- The GQSCAN macro allows you to obtain information about the use of resources.

Figure 6-1 (Page 1 of 2). Task Synchronization Techniques

Pause, Release, and Transfer	**WAIT and POST**
Can change the dispatchability of a task.	Can change the dispatchability of a task.
Can release a task before it is paused.	Can post a task before it waits.
An unauthorized caller can pause and release any task in the caller's home address space.	An unauthorized caller can WAIT and POST any task in the caller's home address space.
A task can only pause on a single PE at a time.	A task may wait on multiple ECBs. If the wait count numbers are posted, the task is made dispatchable.
The Transfer service can simultaneously pause one task and release another.	There is no single service with comparable capability for WAIT and POST.
The Transfer service can release a task and immediately pass control to the released task.	There is no single service with comparable capability for WAIT and POST.

© Copyright IBM Corp. 1988, 1999

Figure 6-1 (Page 2 of 2). Task Synchronization Techniques	
Pause, Release, and Transfer	**WAIT and POST**
The system ensures the Pause Elements are not reused improperly, thus avoiding improper releases caused by unexpected termination or asynchronous ABENDs.	
Ability to pass control directly from one task to another paused task.	
High performance. No local lock contentions effects.	Lower performance, possible local lock contention.
	Callers may use ECBLIST or EVENTS service to wait on multiple ECBs.

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 6-2.

An ECB is also used when the ECB parameter is coded in an ATTACH or ATTACHX macro (see the *OS/390 MVS Programming: Assembler Services Reference* for information on how to use the ATTACH or ATTACHX macro create a new task and indicate the entry point in the program to be given control when the new task becomes active). 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 6-2. 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).

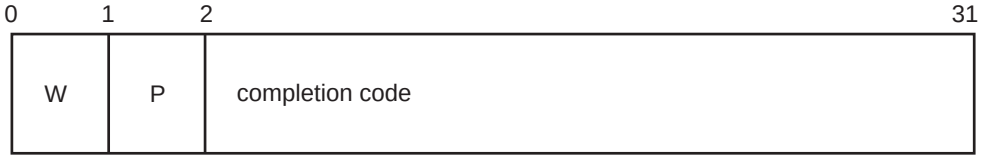


Figure 6-2. Event Control Block (ECB)

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 6-3 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.

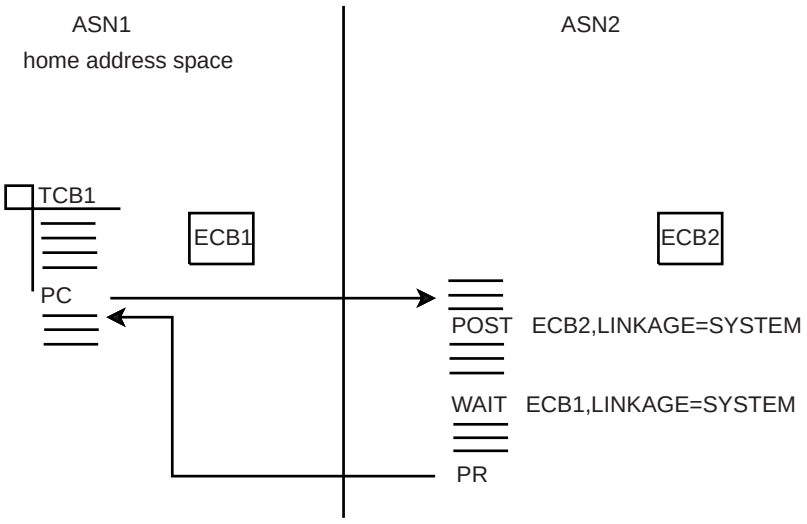


Figure 6-3. Using LINKAGE=SYSTEM on the WAIT and POST Macros

Synchronizing Tasks (Pause, Release, and Transfer)

Pause, Release, and Transfer are callable services that enable you to synchronize task processing with minimal overhead. If you have, for example, an application that requires two or more tasks to trade control back and forth, these services provide efficient transfers of control.

These services, which are available to both unauthorized and authorized callers in Assembler as well as C or C++, use a system-managed object called a pause element to synchronize processing of tasks. The services provide the following functions:

- Pause the current task (Pause service)
- Release a paused task (Release service)
- Simultaneously release a paused task and pass control to it (Transfer service)
- Simultaneously release one paused task and pause the current work unit (Transfer service)

The services use a system-managed pause element (PE) rather than an application-managed control block, such as an event control block (ECB), thus reducing the possibility of error that might come from improper reuse of the control block.

As a PE is much like an ECB, the Pause service is much like the WAIT macro, and the Release service is much like the POST macro. Just as you can use POST to keep a task from waiting by preposting, you can use Release to keep a task from pausing by prereleasing.

The Transfer service can both release a paused task and pass control directly to the released task. The Transfer service can also pause the task that calls the service. Thus, Transfer enables quick dispatches, saving the overhead of work search. It also allows two tasks to trade control back and forth with minimal overhead.

To understand how to use the services, you need to know more about pause elements, (PEs) and the pause element tokens (PETs) that identify them.

Pause Elements and Pause Element Tokens

A pause element (PE) is a system-managed object used to pause and release a task. Like an ECB, a PE is used by the system to control whether or not a task is dispatchable. You can use a PE, like an ECB, to prerelease a task before it is paused. There are, however, significant differences between an ECB and a PE Figure 6-4 compares the two:

Figure 6-4 (Page 1 of 2). Pause Element (PE) and Event Control Block (ECB)

Pause Element (PE)	**Event Control Block (ECB)**
Managed by the system.	Managed by application programs.
Identified by a pause element token (PET).	Identified by a simple address.

Figure 6-4 (Page 2 of 2). Pause Element (PE) and Event Control Block (ECB)

Pause Element (PE)	Event Control Block (ECB)
Cannot be reused once invalidated by an asynchronous ABEND.	Can be reused after a task is removed from the wait state by an asynchronous ABEND, although reuse requires very complicated programming.
To prevent errors related to reuse, a PET can be used only once to identify a PE. Once a task is paused and released, an updated PET is needed to identify the PE. The system returns the updated PET to the caller through the Pause or Transfer service.	There is no control on the reuse of an ECB; the same ECB address can be used over and over.
PEs are allocated and deallocated through calls to system services.	ECBs are allocated by application programs.
The maximum number of PEs available to an unauthorized program is limited; an unauthorized program in a single address space can allocate no more than 1024 PEs at any given time	The system imposes no limits on the number of ECBs.
PEs are not directly addressable through PETs. The user of a PE does not know its actual address and cannot modify it except through the use of system services.	ECBs can be modified at will by application programs.

A PE allocated by an unauthorized caller can be used to pause and release any task in the caller's home address space. An unauthorized caller cannot pause or release using a PE allocated by a fully—authorized caller.

A PE can be used to pause only one task at a time; the system does not allow more then one dispatchable unit to be paused under a single PE.

Using the Services

There are five callable services available for task synchronization:

- Allocate_Pause_Element
- Pause
- Release
- Transfer
- Deallocate_Pause_Element

To use Pause, Release, and Transfer, a program must first allocate a PE by calling the Allocate_Pause_Element service. In response, the system allocates a PE and returns a pause element token (PET) that identifies the pause element (PE).

You use the PET returned from Allocate_Pause_Element to identify the allocated PE until either:

- The PE has been used to pause (through Pause or Transfer) and release (through Release or Transfer) a task.
- A paused task has been released through an asynchronous ABEND.

When you are finished with the PE, call the Deallocate_Pause_Element service to return the PE to the system. If a task is asynchronously ABENDed while it is paused, the system itself invalidates the PE, and it cannot be reused for pause

requests. Thus, return an invalidated PE to the system as soon as possible by a call to Deallocate_Pause_Element.

Though the PE remains allocated until you deallocate it, you can use a PET for only one pair of calls, which result in a pause and a release of a task. When you specify a PET on a successful call to the Pause service or to pause a task through a successful call to the Transfer service, the system invalidates the input PET and returns an updated PET to identify the PE. Use the updated PET to reuse the PE or to deallocate the PE.

Figure 6-5 shows, in pseudocode, the sequence of calls to allocate a PE, pause the current task, release the task, and deallocate the PE.

```
/* Common variables              */

 Dcl PET char(16);

    Workunit #1                          |     Workunit #2

 /* Workunit #1 variables        */     | /* Workunit #2 variables       */
 Dcl Auth1 char(4);                      | Dcl Auth2 char(4);
 Dcl RC1 fixed(32);                      | Dcl RC2 fixed(32;
 Dcl Updated_pet char(16);               | Dcl RelCode binary(24);
 Dcl RetRelCode binary(24);              |
                                         |
 Auth1 = IEA_UNAUTHORIZED;               | Auth2 = IEA_UNAUTHORIZED;
            .                            |            .
            .                            |            .
            .                            |            .
 /* Allocate a Pause Element      */     |
 Call IEAVAPE (RC1,Auth1,PET);           |
                                         |
 /* Pause Workunit #1             */     |
 Call IEAVPSE (RC1,Auth1,PET,            |
            Updated_PET,RetRelCode);     |
                                         |
/*processing pauses until released*/     | RelCode = '123';
                                         | /* Release Workunit #1         */
                                         | Call IEAVRLS (RC2,Auth2,PET,
            .                            |               RelCode);
            .                            |
            .                            |
 PET = UPET;                             |
 Call IEAVPSE (RC1,Auth1,PET);           |
            Updated_PET,RetRelCode);     |
                                         |
/*processing pauses until released*/     | RelCode = '345';
            .                            | /* Release Workunit #1        */
            .                            | Call IEAVRLS (RC2,Auth2,PET,
            .                            |               RelCode);
 /* Deallocate the pause element  */     |
 Call IEAVDPE (RC1,Auth1,                |
            Updated_PET)                 |
```

Figure 6-5. Pause and Release Example

The Pause, Release, and Transfer services also provide a release code field that programs can use to communicate, to indicate, for example, the reason for a release.The program that calls the Release service can set a release code.

The release code is particularly useful when a task might be released before it is paused (prereleased). When a subsequent call to the Pause service occurs, the system does not pause the task; instead, it returns control immediately to the calling program and provides the release code specified on the release call.

Figure 6-6 shows, in pseudocode, the sequence of calls needed to allocate a PE, prerelease a task , and deallocate the PE

```
/* Common variables              */
 Dcl PET char(16);

    Workunit #1

 /* Workunit #1 variables        */
 Dcl Auth1 fixed(32);
 Dcl RC1  fixed(32);
 Dcl Updated_PET char(16);
 Dcl RetRelCode binary(24);

 Auth1 = IEA_UNAUTHORIZED;

 /* Allocate a Pause Element      */
 Call IEAVAPE (RC1,Auth1,PET);
            .
            .
            .

 /* Pause Workunit #1             */
 Call IEAVPSE (RC1,Auth1,PET,
            Updated_PET,RetRelCode);

/*check release code and continue */
            .
            .
            .
 /* Deallocate the pause element  */
 Call IEAVDPE (RC1,Auth1,
            Updated_PET);
```

```
    Workunit #2

/* Workunit #2 variables       */
Dcl Auth2 fixed(32);
Dcl RC2   fixed(32);
Dcl RelCode binary(24);

Auth2 = IEA_UNAUTHORIZED;
RelCode ='123';

/* Release Workunit #1         */
Call IEAVRLS (RC2,Auth2,PET,
              RelCode);
            .
            .
            .
```

Figure 6-6. Release and Pause Example

If you make a release request (through Release or Transfer) specifying a PET that identifies a PE that has not yet been used to pause a task, the system marks the PE as a prereleased PE. If a program tries to pause a task using a prereleased PE, the system returns control immediately to the caller; it does not pause the task. Instead, it resets the PE. As soon as a PE is reset, it can be reused for another Pause and Release, but, as stated earlier, you use the returned updated PET for the next reused PE.

The Pause and Release services are very similar to the WAIT and POST macros, but the Transfer service provides new function. You can use Transfer to either:

- Release a paused task and transfer control directly to the released task
- Pause the current task, release a paused task, and transfer control directly to the released task

Figure 6-7 shows an example of using the Transfer service to release a task without pausing the current task.

Because the Transfer service can affect multiple units of work, using Transfer requires you to work with three PETs:

1. The current pause element token (CurrentDuPet in Figure 6-7) identifies the allocated pause element that Transfer is to use to pause the current task (the caller of the Transfer service). When you do not need to pause the current task , you set this token to binary zeros, as shown in Figure 6-7.
2. The updated pause element token (UPET2 in Figure 6-7), which the system returns when you specify a current pause element token. You need this updated token to reuse the pause element on a subsequent Pause or Transfer or to deallocate the pause element. If you set the current token to binary zeros, as done in Figure 6-7, the contents of the updated pause element token are not meaningful.
3. The target token (TargetDuPET in Figure 6-7) identifies the allocated pause element that Transfer is to use to release a task. In Figure 6-7, it contains the PET that identifies the PE used to pause Workunit #1.

A current release code and a target release code are also available on the call to Transfer. Whether or not each code contains valid data depends on conventions set by the different parts of your program

```
/* Common variables              */
 Dcl PET char(16);

         Workunit #1

/* Workunit #1 variables         */
 Dcl Auth1  char(4);
 Dcl RC1    char(4);
 Dcl UPET1 char(16);
 Dcl RetRelCode binary(24);
           .
           .
           .
Auth1 = IEA_UNAUTHORIZED;
/* Allocate a Pause Element      */
 Call IEAVAPE (RC1,Auth1,PET);

/* Pause Workunit #1             */
 Call IEAVPSE (RC1,Auth1,PET,UPET1,
              RetRelCode);


/*processing pauses until transfer*/




/*processing continues           */

 /* Deallocate the Pause Element */
 Call IEAVDPE (RC1,Auth1,UPET1);
```

```
 Workunit #2

/* Workunit #2 variables        */
Dcl Auth2  char(4);
Dcl RC2   char(4);
Dcl CurrentDuRelCode binary(24);
Dcl CurrentDuPET char(16);
Dcl UPET2 char(16);
Dcl TargetDuPET  char(16);
Dcl TargetDuRelCode char(3);

Auth2 = IEA_UNAUTHORIZED;
           .
           .
           .
TargetDuRelCode = '123';
/* no pause-set token to zeros  */
CurrentDuPet =''B;
TargetDuPET =  PET

/* Transfer to Workunit #1      */
 Call IEAVXFR (RC2,Auth2,
               CurrentDuPET,UPET2,
               CurrentDuRelCode,
               TargetDuPET,
               TargetDuRelCode);
           .
           .
           .
```

Figure 6-7. Transfer without Pause Example

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 a return code indicating the status of the resource.
- Abends the caller on unconditional requests that would otherwise result in 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. If used properly, ENQ/DEQ can 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 unless your program is APF-authorized, or in PSW keys 0-7, or in supervisor state. 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 *OS/390 MVS 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 should 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.”

Limiting Concurrent Requests for Resources

To prevent any one job, started task, or TSO/E 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 abnormally ended 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'.

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 6-8 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 6-8, Step 1) is assigned the resource. The request that established ENTRY2 (Figure 6-8, Step 1) 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.

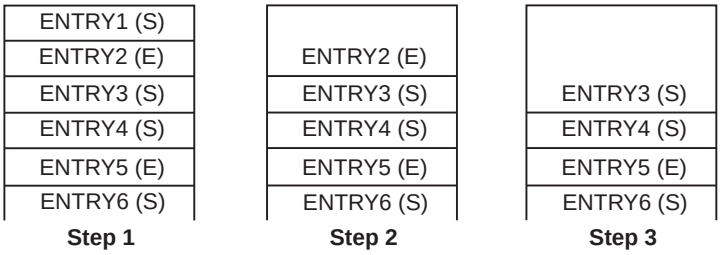


Figure 6-8. ENQ Macro Processing

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 6-8, 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 6-8, 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. With the second request, the system recognizes the contradiction and returns control to the task with a non-zero return code (for a conditional request) or abnormally ends the task (for an unconditional request). You should design your program to ensure that a second request for a resource made by the same task is not issued as an unconditional request 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 the same task does not already have control of or an outstanding request for the same resource.

For the following descriptions, the term “active task” means the task issuing the ENQ macro. No reference is intended to other tasks that 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. 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.

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

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.

The RET=HAVE parameter on DEQ allows you to release control and prevents the control program from abnormally ending the active task if a DEQ attempts to release a resource not held. 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 6-9 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 6-9. 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 6-9 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.
- If you need concurrent use of more than one resource, use the ENQ macro to request control of all such resources at the same 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 6-10, you could code the one ENQ macro shown in Figure 6-11. If all requests were made in this manner, the interlock shown in Figure 6-9 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; multiple resources requested in a single ENQ invocation can be released individually through separate DEQ instructions.

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

Figure 6-10. Two Requests For Two Resources

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

Figure 6-11. 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.

Serializing Access to Resources through the RESERVE Macro

When different systems in your installation access data sets on shared DASD, you can use the ENQ macro with a scope of SYSTEMS to serialize access to those resources. However, you might want to use the RESERVE macro to serialize access when:

- Global resource serialization is not active
- Your installation is not using SMS to manage the shared data sets.

The RESERVE macro provides many of the same functions as ENQ, but can increase contention for resources and the possibility of interlocks. If you use the RESERVE macro to serialize access to data sets on shared DASD, use the DEQ macro to release the resource.

Collecting Information about Resources and Their Requestors (GQSCAN Macro)

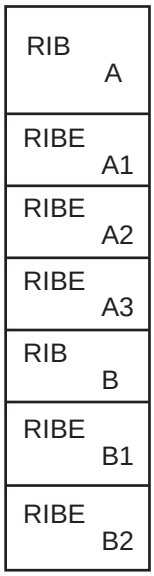
Global resource serialization enables an installation to share symbolically named resources. Programs issue the ENQ and RESERVE macros to request access to resources; global resource serialization adds information about each requestor to the appropriate resource queue. The only way you can extract information from the resource queues is by using the GQSCAN macro.

Using GQSCAN, you can inquire about a particular scope of resources (such as STEP, SYSTEM, or SYSTEMS), a specific resource by name, a specific system's resources, a specific address space's resources, or resources requested through the RESERVE macro. The system collects the information you request from the resource queues and consolidates that information before returning it. The information returned might not reflect changes in the resource queue that occur while the system collects the information.

The system returns the information you request in an area whose location and size you specify using the AREA parameter. The size of the area, the scope of the resource, and whether you code the TOKEN parameter determine the information you receive from GQSCAN. Use the TOKEN parameter when you design your program to issue repeated calls to GQSCAN for the same request. For example, if you request information about a resource that is shared across systems, the amount of information might be more than will fit in the area that you provide. Using the TOKEN parameter allows you to issue subsequent calls to receive additional information about that resource.

How GQSCAN Returns Resource Information

GQSCAN returns the information in the form of resource information blocks (RIB) and resource information block extensions (RIBE), as shown below. The RIB and the RIBE are described in *OS/390 MVS Data Areas, Vol 4 (RD-SRRA)*.



Resource Information block (RIB) describes a resource

RIB extension (RIBE) describes resource requestor

In the RIB, the following fields contain information on RIBEs:

- RIBTRIBE contains the total number of RIBEs associated with this RIB.
- RIBNRIBE contains the total number of RIBEs associated with this RIB that GQSCAN could fit into the area specified on the AREA parameter.
- RIBEDEVN contains a 4-digit EBCDIC device number for RESERVEs issued on the system. For RESERVEs issued on other systems, RIBEDEVN contains zero.

If the value in RIBNRIBE is less than the value in RIBTRIBE, you may need to specify a larger area with the AREA parameter. <! %comment; Moved reference to Data Areas to intro paragraph before RIB graphic Rewrote passages from here to figure 5-2 for CDPI Phase 2 ‘’>

The number of RIBs and RIBEs you receive for a particular resource depends on the size of the area you provide, and the scope and token values you specify on the GQSCAN macro.

How Area Size Determines the Information GQSCAN Returns

The size of the area you provide must be large enough to hold one RIB and at least one of its associated RIBEs; otherwise, you might lose information about resource requestors, or you might have to call GQSCAN repeatedly to receive all of the information you requested. To determine whether you have received all RIBEs for a particular RIB, check the values in the RIBTRIBE and RIBNRIBE fields. To determine whether you have received all of the information on the resource queue, check the return code from GQSCAN.

IBM recommends that you use a minimum area size of 1024 bytes.

The information that GQSCAN returns in the area also depends on what values you specify for the SCOPE and TOKEN parameters.

How Scope and Token Values Determine the Information GQSCAN Returns

Figure 6-12 summarizes the possible values and the information returned for a GQSCAN request.

Figure 6-12. GQSCAN Results with a Scope of STEP, SYSTEM, SYSTEMS, or ALL

GQSCAN Invocation	TOKEN Parameter Coded?	Information Returned
Initial call	No	At least the first RIB that represents the first requestor on the resource queue, and as many of that RIB's associated RIBEs as will fit. Any RIBEs that do not fit are not returned to the caller. If all of the RIBEs fit, GQSCAN returns the next RIB on the resource queue, as long as the remaining area is large enough to hold that RIB and at least one of its RIBEs.
	Yes; value is zero	At least the first RIB that represents the first requestor on the resource queue, and as many of that RIB's associated RIBEs as will fit. Any RIBEs that do not fit are not returned to the caller. If all of the RIBEs fit, GQSCAN returns the next RIB on the resource queue, as long as the remaining area is large enough to hold that RIB and *all* of its RIBEs.
Subsequent call	No	At least the first RIB that represents the first requestor on the resource queue, and as many of that RIB's associated RIBEs as will fit. Any RIBEs that do not fit are not returned to the caller. If all of the RIBEs fit, GQSCAN returns the next RIB on the resource queue, as long as the remaining area is large enough to hold that RIB and at least one of its RIBEs.
	Yes; value is the token value returned by GQSCAN on the preceding call	At least the next RIB on the resource queue, with as many of that RIB's associated RIBEs as will fit. Any RIBEs that do not fit are not returned to the caller. If all of the RIBEs fit, GQSCAN returns the next RIB on the resource queue, as long as the remaining area is large enough to hold that RIB and *all* of its RIBEs.

The example in Figure 6-13 shows the area contents for three requests. For each request, the caller specified the TOKEN parameter and one of the following for the scope value: STEP, SYSTEM, SYSTEMS, or ALL. Assume that the resource queue contains information about four resources: A, which has three requestors; B, which has six; C, which has two; and D, which has one.

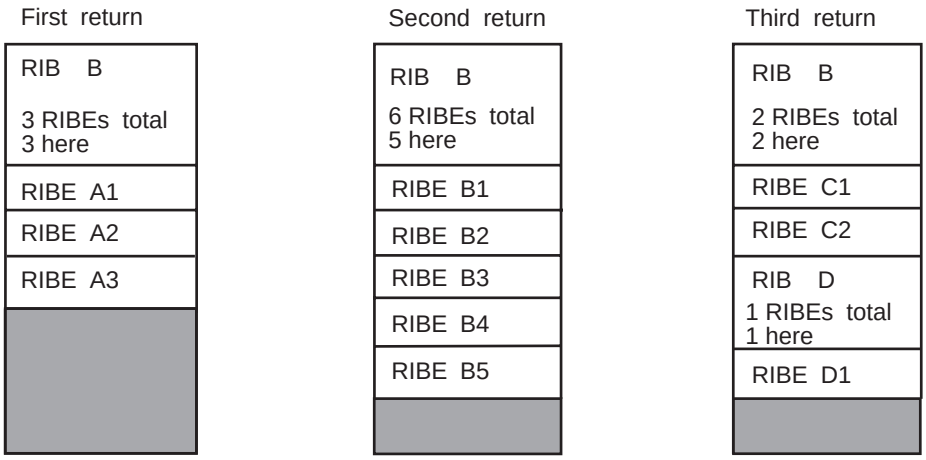


Figure 6-13. Work Area Contents for GQSCAN with a Scope of STEP, SYSTEM, SYSTEMS, or ALL

Note that, because the specified area is not large enough, the caller cannot receive all of the RIBEs associated with resource B, even though the caller coded the TOKEN parameter. To receive all of those RIBEs, the caller has to specify a larger area and reissue the GQSCAN request.

When scanning the information returned, you must use the size of the fixed portion of the RIB and the RIBE that is returned in register 0. The size of the fixed portion of the RIB (RIBLEN) is in the high-order half of register 0, and the size of the RIBE (RIBELEN) is in the low-order half.

The first RIB starts at the beginning of the workarea you specify on the AREA parameter. To find the first RIBE, add the value of RIBLEN and the variable portion of RIB (as found in the RIBVLEN field of the RIB) to the address of the workarea. To find the second RIBE, add the value of RIBELEN to the address of the first RIBE.

To find the second RIB, add the following to the location of the first RIB:

```
RIBLEN + RIBVLEN + (the number of RIBEs × RIBELEN)
```

Chapter 7. Program Interruption Services

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. The program status word's (PSW) program mask provide bits to control certain program exceptions. When MVS gives control to programs, these bits are usually off, disabling the program exceptions. However, MVS also provides the ESPIE and SPIE services to enable program exceptions and to allow a user exit routine to receive control when those exceptions occur. This chapter describes the use of ESPIE and SPIE services.

Specifying User Exit Routines

By issuing the SPIE[1] 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. For the SPIE macro, the exit routine receives control only if the interrupted program is in primary address space control (ASC) mode. For the ESPIE macro, the exit routine receives control if the interrupted program is in either primary or access register (AR) ASC mode. For both the SPIE and ESPIE macros, the exit routine receives control only for interrupts that occur when the primary, home, and secondary address spaces are 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.

[1] The ESPIE macro is the preferred programming interface.

© Copyright IBM Corp. 1988, 1999

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 program interruption control area (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 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 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. See PICA in *OS/390 MVS Data Areas, Vol 3 (IVT-RCWK)* for the mapping provided by the IHAPICA mapping macro.

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 through 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 7-1 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 7-1. 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 program interruption element (PIE) in the virtual storage area assigned to your job step. The system also creates a PIE whenever you issue a SPIE macro and no PIE exists. See PIE in *OS/390 MVS Data Areas, Vol 3 (IVT-RCWK)* for the format of the PIE.

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 (basic control) 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/370 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 system:

- 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 system returns either a token representing the previously active SPIE or ESPIE environment, or a token of zeroes 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 system. 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. ESPIE TEST sets return codes to indicate which type of exit is active, if any, and if one or the other is active, provides information about the exit in a parameter list. Refer to the TEST parameter on the ESPIE macro for a description of the return codes, and the information that is returned in the parameter list.

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 system returns the address of a system-generated PICA in register 1. Do not modify the contents of the system-generated PICA; use the address to restore the previous ESPIE environment.

For a data exception,an ESPIE routine will receive the DXC value in its parameter area, and should use this value rather than the value in the Floating Point Control (FPC) register.

If a retry is to be done, an ESPIE routine can manually change the value(s) of the FPR(s) and FPC register. Changes to the non-volatile fields (i.e., the IEEE settings) in the FPC register must be made carefully since this could affect the processing of the rest of the current program, and possibly subsequent programs.

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 system passes to the ESPIE exit routine when it receives control after a program interrupt. When the exit routine receives control, register 1 contains the address of the EPIE. (See the topic “Environment Upon Entry to User's Exit Routine” for the contents of the other registers.) The format of the EPIE is shown in *OS/390 MVS Data Areas, Vol 2 (DCCB-ITTCTE)*.

Environment Upon Entry to User's Exit Routine

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

Register 0:	Used as a work register by the system.
Register 1:	Address of the PIE or EPIE for the task that caused the interruption.
Registers 2-13:	Unchanged.
Register 14:	Return address.
Register 15:	Address of the exit routine.

The access registers and linkage stack have the values that were current at the time of the program interruption. Both SPIE and ESPIE exits always receive control in primary ASC mode.

Functions Performed in User Exit Routines

Your exit 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 routine can check the two-byte interruption code (the first byte is the exception extension code and the second is the exception code) in the second halfword of the EPIEINT field in the EPIE.
- For a SPIE, your exit 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 exit 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 exit 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 exit routine can alter the contents of general purpose and access registers 0 through 15 in the save area in the EPIE.
- For a SPIE exit, the exit routine can alter general purpose registers 14 through 2 in the register save area in the PIE. To change registers 3 through 13, the exit routine must alter the contents of the registers themselves.

The exit routine can also alter the last four bytes of the OPSW in the PIE or EPIE. For an ESPIE, the exit routine alters the condition code 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.

ESPIE exit routines can alter the ASC mode when control is returned to the interrupted program if the EPIEVERS field in the EPIE contains a value greater than zero. This value is set by the system. To alter the ASC mode of the interrupted program, the exit must do the following:

- Set bit 17 of the EPIEPSW field in the EPIE. If this bit is 0 when control is returned to the interrupted program, the program receives control in primary ASC mode. If this bit is 1 when control is returned to the interrupted program, the program receives control in AR ASC mode.
- Set the EPIERCTL bit in the EPIE to indicate that the ASC mode for the interrupted program has been set by the exit routine.

Chapter 8. Providing Recovery

In an ideal world, the programs you write would run perfectly, and never encounter an error, either software or hardware. In the real world, programs do encounter errors that can result in the premature end of the program's processing. These errors could be caused by something your program does, or they could be beyond your program's control.

MVS allows you to provide something called **recovery** for your programs; that means you can anticipate and possibly recover from software errors that could prematurely end a program. To recover from these errors, you must have one or more user-written routines called **recovery routines**. The general idea is that, when something goes wrong, you have a recovery routine standing by to take over, fix the problem, and return control to your program so that processing can complete normally; if the problem cannot be fixed, the recovery routine would provide information about what went wrong. If correctly set up, your recovery should, at the very least, provide you with more information about what went wrong with your program than you would have had otherwise.

Part of recovery is also the "cleaning up" of any resources your program might have acquired. By "clean up" of resources, we mean that programs might need to release storage that was obtained, release ENQs, close data sets, and so on. If your program encounters an error before it has the opportunity to clean up resources, your recovery routine can do the clean up.

MVS provides the recovery termination manager (RTM) to handle the process by which recovery routines and resource managers receive control.

This chapter is devoted to explaining why you might want to provide recovery for your programs in anticipation of encountering one or more errors, and how you go about doing that. An important point to note is that providing recovery is something to be considered **at the design stage of your program**. You should make the decision about whether to provide recovery before you begin designing the program. Trying to provide recovery for an existing program is much more difficult because recovery must be an integral part of your program.

The following table provides a roadmap to the information in this chapter. If you already understand recovery concepts you might want to skip directly to those sections of specific interest to you.

© Copyright IBM Corp. 1988, 1999

To find out about:	Consult the following section:
General recovery concepts, including: • Why you would want to provide recovery. • What software errors result in your recovery getting control. • What we mean when we say a program **abnormally ends**. • The different states for a recovery routine. • The different types of routines in a recovery environment, and how to choose, define, and activate the right recovery routine. • The basic options available to a recovery routine. • How routines in a recovery environment interact.	"Understanding General Recovery Concepts" on page 8-2.
How to write a recovery routine, including: • What recovery routines do. • How recovery routines communicate with other routines and with the system. • Special considerations when writing different types of recovery routines.	"Writing Recovery Routines" on page 8-11.
The recovery environment, including: • Register contents at various times during recovery processing. • Other environmental factors such as program authorization, dispatchable unit mode, ASC mode, and so on.	"Understanding the Recovery Environment" on page 8-30.
Coding the various routines in a typical recovery environment.	"Understanding Recovery through a Coded Example" on page 8-42.
Advanced recovery topics, including: • Intentionally invoking RTM. • Providing multiple recovery routines. • Providing recovery for recovery routines. • Providing recovery for multitasking programs	"Understanding Advanced Recovery Topics" on page 8-45.

Understanding General Recovery Concepts

This section provides a general overview of recovery concepts. After reading this section, you should understand the following:

- Why you would want to provide recovery for your programs.
- What software errors result in your recovery getting control, if you provide recovery.
- What we mean when we say a program **abnormally ends**.
- The different **states** for a recovery routine.

- The difference between a **mainline routine**, a **recovery routine**, and a **retry routine**.
- What an **extended specify task abnormal exit (ESTAE-type)recovery routine** is and how to choose, define, and activate the appropriate one.
- The difference between what it means to **retry** and what it means to **percolate**.
- How routines in a recovery environment interact.

Deciding Whether to Provide Recovery

MVS does all that it can to ensure the availability of programs, and to protect the integrity of system resources. However, MVS cannot provide effective recovery for every individual application program, so programs need recovery routines of their own.

To decide whether you need to provide recovery for a particular program, and the amount of recovery to provide, you should:

- Determine what the consequences will be if the program encounters an error and ends.
- Compare the cost of tolerating those consequences to the cost of providing recovery.

In general, if you have a large, complex program upon which a great number of users depend, such as a subsystem, a database manager, or any application that provides an important service to many other programs or end users, you will almost certainly want to provide recovery. For small, simple programs upon which very few users depend, you might not get enough return on your investment. Between these two extremes is a whole spectrum of possibilities.

Consider the following points in making your decision. **Providing recovery**:

- **Increases your program's availability**.

  Depending on the nature of the error, your recovery routine might successfully correct the error and allow your program to continue processing normally. Maintaining maximum availability is one of the major objectives of providing recovery.

- **Is a way to protect both system and application resources**.

  In general, recovery routines should clean up any resources your program is holding that might be requested by another program, or another user of your program. The purpose of clean up is to:

  – Allow your program to run again successfully without requiring a re-IPL
  – Allow the system to continue to run other work (consider especially other work related to the failing program).

  Virtual storage and ENQs are examples of important resources shared by other programs. A program should provide for the release of these resources if an error occurs so that other programs can access them.

  Note: ENQs are used for serialization. See Chapter 6, “Resource Control” on page 6-1 for more information about serialization.

  Recovery routines should also ensure the integrity of any data being accessed. Consider the case of a database application that is responsible for protecting its

database resources. The application must ensure the integrity and consistency of the data in the event an error occurs. Data changes that were made prior to the error might have to be backed out from the database.

- **Provides for communication between different processes.**

  An example of this would be a task that sends a request to another task. If the second task encounters an error, a recovery routine could inform the first task that its request will not be fulfilled.

  When dealing with a multi-tasking environment, you must plan your recovery in terms of the multiple tasks involved. You must have a cohesive scheme that provides recovery for the set of tasks rather than thinking only in terms of a single task.

- **Is a way to help you determine what went wrong when an error occurs in your program.**

  Recovery routines can do such things as save serviceability data and request dumps to help determine what went wrong in your program. These actions are explained in greater detail later in this chapter.

- **Facilitates validity checking of user parameters.**

  Consider the case of a program that must verify input from its callers. The program does parameter validation, but might not catch all variations. For example, the caller might pass the address of an input data area that appears to be valid; however, the caller did not have access to that storage. When the program attempts to update the data area, a protection exception occurs. A recovery routine could intercept this error, and allow the program to pass back a return code to the caller indicating the input was not valid.

  Providing recovery in a case like this improves the reliability of your program.

If you **do not** provide recovery for your program, and your program encounters an error, MVS handles the problem to some extent, but the result is that your program ends before you expected it to, and application resources might not be cleaned up.

Understanding Errors in MVS

Certain errors, which your program or the system can detect, trigger the system to interrupt your program and pass control to your recovery routine (or routines) if you have any; if you do not have any recovery routines, the system **abnormally ends** your program. This chapter uses the term **abnormal end** when your program ends for either of the following reasons:

- Your program encounters an error for which it has no recovery routines
- Your program encounters an error for which its recovery routines are not successful.

The errors for which you, or the system, might want to interrupt your program are generally those that might degrade the system or destroy data.

The following are some examples of errors that would cause your recovery routine (if you have one) to get control:

- Unanticipated program checks (except those resolved by SPIE or ESPIE routines; see Chapter 7, “Program Interruption Services” on page 7-1 for information about SPIE and ESPIE routines.)

- Machine checks (such as a storage error that occurs while your program is running)
- Various types of CANCEL (such as operator or time out)
- An error when issuing an MVS macro or callable service (for example, specifying parameters that are not valid)

Each of the above errors has associated with it one or more **system completion codes**. All system completion codes are described in *OS/390 MVS System Codes*. You can write your recovery routine to specifically handle one or more of these system completion codes, or define your own user completion codes and handle one or more of them. Completion codes associated with errors are also referred to as **abend codes**.

As stated earlier, the system can detect errors, but your program also can detect errors and request that the system pass control to recovery routines. To do so, your program can issue the ABEND macro.

Use the ABEND macro to request recovery processing on behalf of the current unit of work. Your program might choose to issue the ABEND macro if it detects an impossible or illogical situation and cannot proceed further. For example, your program might be passed parameters that are not valid, or might detect something in the environment that is not valid. Your program might also choose to issue the ABEND macro so that its recovery routine can get control to save serviceability information.

Understanding Recovery Routine States

In this chapter we talk about recovery routines being in one of the following states:

- **Defined**

  A recovery routine is **defined** when you make it known to the system. For example, you might issue a macro on which you specify a particular recovery routine. At the point of issuing that macro, the recovery routine is defined to the system.

- **Activated**

  A recovery routine is **activated** when it is available to receive control; if an error occurs, the system can pass control to an activated recovery routine. Depending on the type of recovery routine, it might be defined to the system but not yet activated. Some recovery routines are both defined and activated by issuing a single macro.

- **In control**

  A recovery routine is **in control** when it is running; an error has occurred and the system passed control to the recovery routine.

- **No longer in control**

  A recovery routine is **no longer in control** when it returns control to the system. The recovery routine returns control either by requesting to percolate or retry (terms defined later in this chapter) and issuing a BR 14 instruction, or by encountering an error itself.

- **Deactivated**

A recovery routine is **deactivated** when it is no longer available to receive control; if an error occurs, the system **will not** pass control to a deactivated recovery routine. Depending on the type of recovery routine, it might be deactivated but still defined to the system. For some recovery routines, issuing a single macro results in the routine becoming both deactivated and no longer defined.

- **No longer defined**

  A recovery routine is **no longer defined** when it is no longer known to the system. The routine might still exist and be in virtual storage, but the system no longer recognizes it as a recovery routine.

Understanding the Various Routines in a Recovery Environment

This chapter discusses the following different types of routines that interact in a recovery environment:

- Mainline routine
- Recovery routine
- Retry routine (also known as a retry point)

All of these routines are user-written routines. The following section provides you with a description of each of these routines.

Mainline Routine

The mainline routine is that portion of your program that does the work, or provides the required function. In general, the mainline routine defines and activates the recovery routine. Before returning to its caller, the mainline should also deactivate the recovery routine and request that it be no longer defined. When an error occurs in the mainline routine, the system passes control to the recovery routine.

Recovery Routine

A recovery routine is the routine to which the system passes control when an error occurs in the mainline routine. The recovery routine's objective is to intercept the error and potentially perform one or more of the following tasks:

- Eliminate or minimize the effects of the error
- Allow the mainline routine to resume normal processing
- Clean up resources
- Communicate with other programs as appropriate
- Provide serviceability data
- Request a dump
- Validate user parameters
- Provide one or more recovery routines for itself.

The recovery routine can be an entry point in your program that processes only when an error occurs, or it can be a separate routine that gets control when the error occurs.

Retry Routine

A retry routine is essentially an extension of the mainline routine. When an error occurs, the system passes control to your recovery routine, which can then request the system to pass control back to the mainline routine to resume processing. That portion of the mainline that gets control back is referred to as the retry routine. When the retry routine gets control, it is as if the mainline routine branched there

after encountering the error; to the mainline routine, it appears as though the error never occurred.

The retry routine does whatever processing your mainline routine would continue doing at that point.

Once the retry routine is running, if another error occurs, the system again passes control to your recovery routine, just as it did when the mainline routine encountered an error.

Choosing the Appropriate Recovery Routine

The recovery routines you can provide are called **ESTAE-type recovery routines**. This section describes the different types of ESTAE-type recovery routines, and for each type, describes how you define it, activate it, deactivate it, and request that it no longer be defined. A summary of this information is in Figure 8-1 on page 8-9.

When you provide one or more recovery routines for your program, you have the opportunity to identify a user parameter area for the system to pass from the mainline routine to the recovery routine. Creating such a parameter area with information for the recovery routine is a very important part of providing recovery. See “Setting Up, Passing, and Accessing the Parameter Area” on page 8-19 for more information about what this parameter area should contain, and how to pass it.

Define ESTAE-type recovery routines in the following ways:

- STAE, ESTAE, and ESTAEX macros
- ATTACH and ATTACHX macros with STAI and ESTAI parameters

The following describes the recovery routines you can define with each of the above macros:

- **STAE, ESTAE, and ESTAEX macros**

  To provide recovery to protect itself and any other programs running under the same task, a program can issue either the STAE, ESTAE, or ESTAEX macro with the CT parameter. Each of these macros both defines and activates the recovery routine. The recovery routine is defined and activated until one of the following events occurs:

  – You deactivate it and request that it be no longer defined (issue STAE 0, ESTAE 0, or ESTAEX 0).

  – The recovery routine fails to or chooses not to retry (explained under “Understanding Recovery Routine Options” on page 8-9).

  – The request block (RB) under which the caller of the macro is running terminates.

  A program cannot protect other tasks with recovery routines defined through these macros.

  IBM recommends you always use ESTAEX unless your program and your recovery routine are in 24-bit addressing mode, in which case, you should use ESTAE. ESTAE and ESTAEX provide the same function, except that ESTAEX can be issued in AR ASC mode. STAE is the pre-MVS/XA version of ESTAE.

The remainder of this chapter refers to the recovery routines you define and activate through the ESTAE and ESTAEX macros as **ESTAE routines** or **ESTAEX routines**, respectively.

- **ATTACH and ATTACHX macros with STAI and ESTAI parameters**

  To attach a task and provide recovery to protect the attached task and all of its subtasks, a program can issue either the ATTACH or the ATTACHX macro with either the STAI or the ESTAI parameter. You define the recovery routine when you issue the macro. The recovery routine is not activated until the attached task gets control. The recovery routine remains activated as long as the attached task is still running, or until the recovery routine fails to or chooses not to retry. The system deactivates the recovery routine when the attached task ends. At that point, the recovery routine is no longer defined.

  The program attaching the task is not protected by the recovery defined in this manner. Only the attached task and its subtasks are protected.

  IBM recommends you always use the ESTAI, rather than the STAI, parameter on ATTACHX, rather than ATTACH. ATTACH and ATTACHX provide the same function, except that ATTACHX can be issued in AR ASC mode. STAI is the pre-MVS/XA version of ESTAI.

  The remainder of this chapter refers to the recovery routines you define through ATTACHX with ESTAI as **ESTAI routines**. All references to the ATTACHX macro apply also to the ATTACH macro.

  In summary, **ESTAE-type recovery routines** include **ESTAE and ESTAEX routines**, and **ESTAI routines**.

  Note: Another ESTAE-type recovery routine that you can write is the associated recovery routine (ARR). ARRs are associated with stacking PC routines. Only authorized programs can create the environment required to use these routines. For information about coding ARRs, see the application development books that are available to the programmers in your installation that use authorized macros.

Floating Point Implications

When working under the FRR recovery routine state, the first recovery routine will normally see the time-of-error Floating Point Registers (FPRs) and the Floating Point Control (FPC) register. The DXC value is provided in the SDWA. It is this value that should be used rather than the copy in the Floating Point Control register.

If control can pass to other recovery routines, and the first recovery routine modifies any of the FPRs or FPC register, it is responsible to save and restore the time-of-error FPRs and FPC register.

If retry is to be done, a recovery routine can (manually) change the value(s) of the FPR(s) and FPC register. Changes to the non-volatile fields (i.e., the IEEE settings) in the FPC register must be made carefully since this could affect the processing of the rest of the current program, and possibly subsequent programs.

Summary of Recovery Routine States

The following table summarizes, for each type of recovery routine, when the recovery routine is defined, activated, deactivated, and no longer defined.

Figure 8-1. Summary of Recovery Routine States

Recovery routine	Defined	Activated	Deactivated	No longer defined
ESTAE	ESTAE CT	ESTAE CT	ESTAE 0	ESTAE 0
ESTAEX	ESTAEX CT	ESTAEX CT	ESTAEX 0	ESTAEX 0
ESTAI	ATTACHX ESTAI	Attached task gets control	Attached task ends	Attached task ends

Understanding Recovery Routine Options

A recovery routine has two basic options: the routine can either **retry** or it can **percolate**.

Retry is the attempt to resume processing at some point in the unit of work that encountered the error. The recovery routine does something to circumvent or repair the error, and requests that the system pass control to a retry routine to attempt to continue with normal processing.

Percolate is the opposite of **retry**. To percolate means to continue with error processing. A recovery routine percolates under one of the following circumstances:

- The system does not allow a retry
- The recovery routine chooses not to retry, perhaps because the environment is so damaged that the routine cannot circumvent or repair the error, or perhaps because the recovery routine was designed only to capture serviceability data, and is not intended to retry.

When a recovery routine percolates, the system checks to see if any other recovery routines are activated. If so, the system passes control to that recovery routine, which then has the option to either retry or percolate. Think of the process of percolation, then, as the system passing control to one recovery routine after another.

The system gives control to ESTAE-type recovery routines in the following order:

1. ESTAE-type recovery routines that are not ESTAI routines, in last-in-first-out (LIFO) order, which means the most recently activated routine gets control first
2. ESTAI routines, in LIFO order.

Once all routines have percolated, the system proceeds to abnormally end your program. See “Providing Multiple Recovery Routines” on page 8-46 for more information about having multiple recovery routines.

Understanding How Routines in a Recovery Environment Interact

Figure 8-2 on page 8-10 is a very simplified illustration of how routines in a recovery environment interact. In this figure, only one recovery routine exists, and it is an ESTAE-type recovery routine. The following sequence of events might occur:

1. The mainline routine encounters an error.

2. The system gets control.
3. The system looks for recovery routines and finds an ESTAE-type recovery routine called ESTAEX.
4. The ESTAEX routine either retries or percolates.
   a. If the ESTAEX routine retries, it returns control to a retry point in the mainline routine. The mainline routine continues processing.
   b. If the ESTAEX routine percolates, the system gets control and abnormally ends the mainline routine.

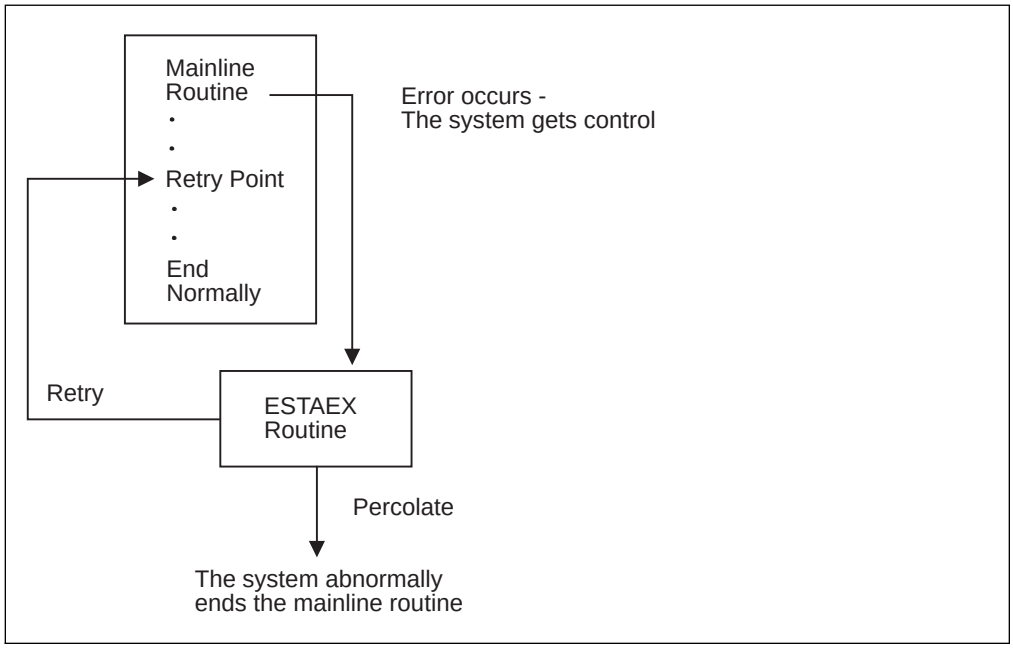


Figure 8-2. Mainline Routine with One Recovery Routine

Figure 8-3 on page 8-11 shows a more complex situation. Several recovery routines exist, and each one that is entered has the opportunity to retry or to percolate. The following sequence of events might occur if all recovery routines percolate:

1. The mainline routine encounters an error.
2. The system looks for recovery routines, and finds that ESTAEX(3) was the last one created.
3. The system gives control to ESTAEX(3) first.
4. ESTAEX(3) percolates to ESTAE(2), which percolates to ESTAI(1).
5. ESTAI(1) also percolates, and no other recovery routines are activated, so the system abnormally ends the mainline routine.

Had any of the recovery routines decided to retry, the system would have returned control to the retry point, and the mainline routine might have ended normally.

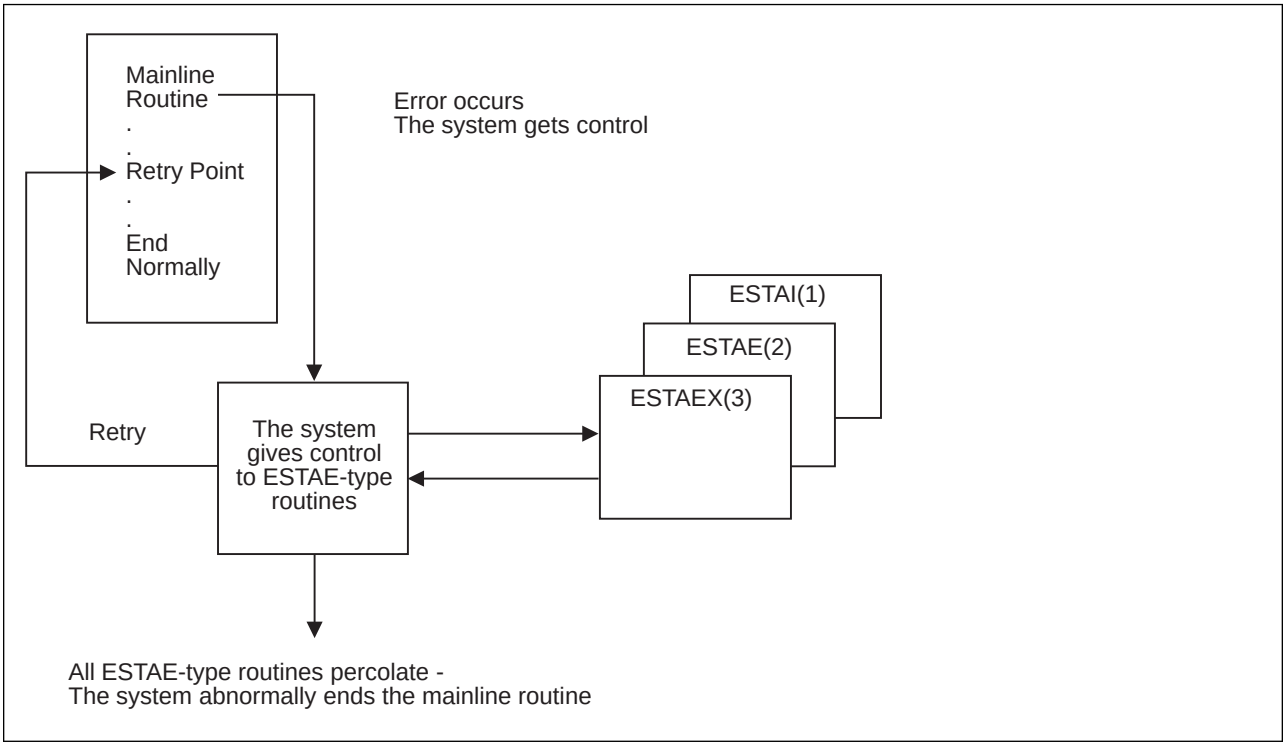


Figure 8-3. Mainline Routine with Several Recovery Routines

Writing Recovery Routines

So far, this chapter has discussed general recovery concepts, including how to decide what type of recovery you need, and how to provide that recovery. But you have to write the recovery routines that you provide. To do so, you must understand all of the following. Each item is described in greater detail in the sections that follow.

- **What a recovery routine is supposed to do.**

  So far we talked about how recovery routines can either retry or percolate. But, they do a lot more than that. We also talked about recovery routines correcting or repairing errors, but we have not said how exactly they go about doing that.

- **How the recovery routine communicates with the mainline routine, the retry routine, and the system.**

  The means of communication available to a recovery routine are:

  – A user parameter area, built by the mainline routine and passed to the recovery routine.

  – A data area called the system diagnostic work area (SDWA), which is provided by the system. The recovery routine communicates with the system, with other recovery routines, and with the retry routine through the SDWA. The recovery routine uses the SETRP macro to update information in the SDWA.

  – Registers, when no SDWA is provided.

- **The special considerations you must make when writing an ESTAE-type recovery routine.**

  One important consideration is the presence of an SDWA. The case where an SDWA is not provided is rare; nevertheless, when you design an ESTAE-type recovery routine, you must allow for the possibility of not receiving an SDWA.

  Special considerations for ESTAE-type recovery routines also include RB considerations, linkage stack considerations, and outstanding I/Os at time of failure.

Note: When an error occurs for which the system passes control to your recovery routine, the recovery routine must be in virtual storage. It can either be an entry point in your program, or a separate routine. You are responsible for ensuring that the recovery routine is in virtual storage when needed.

Understanding What Recovery Routines Do

The following is a list of some of the things a recovery routine should do if the recovery is to be effective. Each item is explained in greater detail in the sections that follow.

The items are arranged in a way that suggests the order in which you might do them; however, you must decide yourself the order that would work best for your particular routine.

- Preserve the return address to the system.
- Check for the presence of an SDWA.
- Establish addressability to the parameter area passed by the mainline routine. How you do that depends on whether an SDWA is present.
- Check the contents of important fields in the SDWA.
  - Determine the location of the parameter area.
  - Determine why the routine was entered.
  - Determine if this is the first recovery routine to get control.
- Check the contents of the parameter area passed by the mainline.
  - Determine if this is a repeated error (to avoid recursion).
  - Determine when and where the error occurred.
- Provide information to help determine the cause of the error:
  - Save serviceability data in the SDWA.
  - Request a dump of storage.
- Try to correct or minimize the effects of the error.
- Determine whether the recovery routine can retry, decide whether to retry or percolate, and take the appropriate actions (such as cleaning up resources).

Saving the Return Address to the System

When writing a recovery routine, you must save the return address to the system, which you find in general purpose register (GPR) 14. The system sets up the return address so that the recovery routine can return, at the appropriate time, using a BR 14 instruction.

Checking for the SDWA

For an ESTAE-type recovery routine, if the system cannot obtain storage for an SDWA, the system does not provide one. The case where an SDWA is not provided is rare. Nevertheless, when you design an ESTAE-type recovery routine, you must allow for the possibility of not receiving an SDWA; almost every action an ESTAE-type recovery routine takes must be set up differently to handle the two possibilities.

To check for the presence of the SDWA, the recovery routine checks the contents of GPR 0. If GPR 0 contains 12 (X'0C') the system **could not** obtain an SDWA. When GPR 0 contains any value other than 12, an SDWA is present, and its address is in GPR 1. When the system provides an SDWA, the system also provides a register save area whose address is in GPR 13.

If an SDWA was not provided GPR 13 does not point to a save area, and your routine must not use the area pointed to by GPR 13.

Establishing Addressability to the Parameter Area

The recovery routine also must establish addressability to the parameter area passed by the mainline routine. To determine the location of the parameter area:

- If an SDWA is present, the recovery routine checks either the contents of SDWAPARM or the contents of GPR/AR 2. GPR 2 contains the address of the parameter area, and for AR-mode callers, AR 2 contains the ALET.
- If no SDWA is present, the recovery routine checks the contents of GPR/AR 2. GPR 2 contains the address of the parameter area, and for AR-mode callers, AR 2 contains the ALET.

The following are examples of information a mainline routine can pass to a recovery routine through the parameter area:

- A dynamic storage area
- An input parameter list (that is, a parameter list that might have been passed to the mainline routine)
- The addresses of important data areas.

Checking Important Fields in the SDWA

Assuming an SDWA is present, your routine can obtain a great deal of information from this data area. Some of the key information a recovery routine can check for in the SDWA includes:

- **Why the routine was entered.**

  The routine can check the SDWACMPC field, which contains the completion code that existed when the system gave control to the routine, and the SDWACRC field, which contains the reason code associated with the completion code. SDWACRC contains a reason code only if the SDWARCF bit is on.

- **The location of the parameter area that was passed by the mainline.**

  The routine can check the SDWAPARM field, which provides the information the routine needs to locate the parameter area. The contents of this field vary depending on the way in which the recovery was defined.

- **Whether this is the first recovery routine to get control.**

If the SDWAPERC bit is off, this recovery routine is the first to get control. If the SDWAPERC bit is on, percolation has occurred.

The first recovery routine to get control usually has a more direct relationship with the error; being the first recovery routine to get control for an error can be an indication that the error occurred in the mainline routine that activated this particular recovery routine, rather than in a routine that was subsequently called.

This information can be useful in determining what action the recovery routine should take. A recovery routine is more likely to take corrective action or capture serviceability data if it is the first to get control for an error. Subsequent recovery routines are further removed from the error, and might limit their activities to releasing resources, or attempting a retry if possible.

See “Important Fields in the SDWA” on page 8-23 for a list of some of the fields in the SDWA, and an explanation of their contents.

Checking the Contents of the Parameter Area

Generally the mainline routine sets up a parameter area containing information for use by the recovery routine. Key information that a recovery routine might determine from the parameter area includes:

- When and where the error occurred
- Whether this is a repeated error.

The recovery routine can tell when and where the error occurred through “footprints,” a technique explained under “Deciding What to Include in the Parameter Area” on page 8-19. Footprints can help the recovery routine to avoid getting into a loop in which the routine requests a retry, and the same error occurs again (recursion). For example, if the recovery routine supplies a bad retry address to the system, and the processing 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 created.

See “Setting Up, Passing, and Accessing the Parameter Area” on page 8-19 for more information about what the parameter area can contain, and the techniques you can use to provide the most useful information to the recovery routine.

Saving Serviceability Data

One of the objectives of providing recovery is to obtain as much information as possible to help you determine what went wrong. The SDWA has certain areas where the recovery routine can save such information. Your recovery routine can update the SDWA with serviceability information in three different ways:

- By issuing the SETRP macro with the RECPARM parameter. Use the RECPARM parameter to supply the load module name, the active CSECT name, and the recovery routine CSECT name. See “Using the SETRP Macro to Update the SDWA” on page 8-22 for more information about using SETRP.
- By issuing the VRADATA macro to update the SDWA variable recording area. See the VRADATA macro in *OS/390 MVS Programming: Assembler Services Reference* for more information.

- By directly manipulating other fields in the SDWA. Important fields to fill in are SDWACID, SDWASC, SDWAMLVL, and SDWARRL. See “Important Fields in the SDWA” on page 8-23 for a description of each of these fields.

Part of saving serviceability data includes providing information for dump analysis and elimination (DAE). DAE depends on information that users provide in recovery routines to construct symptom strings needed to describe software failures. DAE uses these symptom strings to analyze dumps and suppress duplicate dumps as requested. You should provide information for DAE prior to requesting a dump of storage. See “Suppressing Dumps That Duplicate Previous Dumps” on page 9-2 for more information about DAE and dump suppression.

Requesting a Dump

Your recovery routine can also request a dump of storage to help determine the cause of the error. In most cases, the system does not automatically request dumps on behalf of your program. To request an ABEND dump, the recovery routine can issue the SETRP macro with the DUMP=YES parameter.

For more information about requesting dumps, see Chapter 9, “Dumping Virtual Storage (ABEND, SNAPX, SNAP, and IEATDUMP Macros)” on page 9-1.

Before requesting a dump of storage, the recovery routine should check the SDWAEAS bit. The SDWAEAS bit is on when a previous recovery routine has provided sufficient diagnostic data related to this error. The recovery routine that analyzes the problem and captures sufficient diagnostic data is responsible for setting the SDWAEAS bit so that subsequent recovery routines know they do not have to capture any further data.

Note that if your program calls a system service (by issuing a macro or callable service), that system service might encounter a user-induced error and end abnormally. Generally, the system does not take dumps for user-induced errors. If you require such a dump, then it is your responsibility to request one in your recovery routine.

Correcting or Minimizing the Error

Another important activity for a recovery routine is to attempt to correct or minimize the error. What the recovery routine actually does to correct or minimize the error depends on what the mainline routine is doing and what the error is. Some examples of possible situations where the recovery routine could take action are the following:

- The mainline routine might be working with a queue of data areas. The recovery routine might be able to scan the queue and determine if one or more of the data areas contains information that is not valid.

  For example, one of the data areas might contain an address that is not valid. Or, the mainline routine might have set up the data areas with some sort of validating information that could be checked, and possibly corrected. Certain data areas might have to be deleted from the queue, or the entire queue might have to be deleted and rebuilt.

- The mainline routine might be running under a task that is communicating with another task when an error occurs. The recovery routine might then take the action of alerting the other task that a problem exists, so the other task does not wait for any further communication.

- The mainline routine might have initiated I/O, and the recovery routine might have to ensure that the I/O completes to protect the integrity of the I/O resources.
- The recovery routine might back out changes made to a database to ensure its integrity.

Deciding to Retry or Percolate

Under certain circumstances (such as CANCEL), the system does not allow a retry. The SDWACLUP bit is on when the system prohibits a retry, and off when the system allows a retry.

If a recovery routine requests retry when it is not allowed, the system ignores the request and continues with percolation.

A recovery routine must determine whether it will attempt a retry. The determination might be very simple: if the SDWACLUP bit is on, retry is not even an option. But if retry **is** an option, the routine must make the decision based on the information it has gathered in the preceding steps.

By no means is a recovery routine required to attempt a retry, even when one is permitted. The recovery routine might decide not to retry if no SDWA is present, going on the assumption that serious problems probably exist. The routine might make the decision based on the particular completion code it finds in SDWACMPC, or based on information in the parameter area, or based on how successful the routine was in determining the cause of the error and fixing it. Perhaps the environment is so badly damaged that repair is beyond the scope of the recovery routine.

Once the decision is made, the recovery routine now does different things depending on whether it will retry or percolate.

Note: If the recovery routine does not specify retry or percolate, the default is to percolate.

Recovery Routines that Retry: When a recovery routine decides to retry, it should do the following:

- Eliminate or minimize the cause of the error with complete or partial repair, as explained above under “Correcting or Minimizing the Error” on page 8-15.
- Ensure that the retry routine's environment is restored. For example, restore registers and re-establish addressability to mainline resources. See “Register Contents on Entry to a Retry Routine” on page 8-35 for details about how a recovery routine can control the register contents on entry to the retry routine.
- Know the condition of resources being held by the mainline. For example, the routine might have to repair data structures, back out changes to data sets, and so on.
- Indicate to the system that a retry is to be attempted. If an SDWA is present, the recovery routine issues the SETRP macro with the RC=4 parameter to indicate retry, and the RETADDR parameter to specify the address of the retry routine. You can specify RC=4 even when the SDWACLUP bit is on, indicating that retry is not allowed. If you do so, however, the system ignores the retry request.

If no SDWA is present, the recovery routine has to set a return code of 4 in GPR 15, and place the address of the retry routine in GPR 0.

- Decide whether to pass the SDWA to the retry routine, and so indicate on the SETRP macro with the FRESDWA parameter.

What the Retry Routine Does: Once the retry routine gets control, it continues with mainline processing, and can free resources, deactivate recovery routines, and so on. As stated earlier, the retry routine is really an extension of the mainline routine, and its purpose is to re-establish the mainline environment.

When the retry routine gets control, the following are true:

- The retry routine runs under the same unit of work that activated the recovery routine. See “Special Considerations for ESTAE-Type Recovery Routines” on page 8-26 for further details related to ESTAE-type recovery routines.
- The retry routine might or might not have access to the SDWA, and the recovery routine might or might not have directed that register contents be restored for the retry routine.

  For ESTAE-type recovery routines that specify FRESDWA=YES on SETRP, the system frees the SDWA before entering the retry routine. For ESTAE-type recovery routines that specify RETREGS=YES, the system restores the registers from the SDWA.

  For ESTAE-type recovery routines that specify FRESDWA=NO on SETRP, the system does not free the SDWA, and the retry routine can access it. **In that case, the retry routine also has the responsibility of freeing the storage for the SDWA when it is no longer needed.** The subpool number and length to use to free the storage are in the SDWA, in fields SDWASPID and SDWALNTH, respectively.

  Note: **IBM recommends** that the recovery routine use FRESDWA=YES on the SETRP macro, thus alleviating the retry routine's responsibility to free the SDWA. If your recovery routine retries multiple times and the SDWA is not freed, out-of-storage failures can result.

  The retry routine can determine what action the recovery routine took in regard to freeing the SDWA and restoring registers by examining the contents of GPR 0:

Figure 8-4. Contents of GPR 0 on Entry to a Retry Routine	
GPR 0 Contents	**Meaning**
0	The system provided an SDWA. The recovery routine specified RETREGS=NO and FRESDWA=NO. Registers are not restored from the SDWA, and the retry routine must free the SDWA. GPR 1 contains the address of the SDWA.
12 (X'0C')	The system did not provide an SDWA.
20 (X'14')	The system provided an SDWA. The recovery routine specified RETREGS=NO and FRESDWA=YES. Registers are not restored from the SDWA, and the retry routine does not have to free the SDWA.
Value restored from SDWA (field SDWASR00)	The system provided an SDWA. The recovery routine specified RETREGS=YES, and either FRESDWA=NO or FRESDWA=YES. If the recovery routine specifies FRESDWA=NO, the recovery routine must alert the retry routine to free the SDWA. Some sort of protocol must be established between the recovery routine and the retry routine. For example, the recovery routine can set a unique value in SDWASR00 (the field that represents GPR 0 in SDWASRSV) to distinguish this case from those above where GPR 0 contains either 0, 12, or 20. The recovery routine can pass the address of the SDWA to the retry routine in a parameter area (use the parameter area pointed to by SDWAPARM) or in a register (consider using register 0).

For complete details about register contents see “Understanding the Recovery Environment” on page 8-30.

- The recovery routine that requested the retry is still activated and can be entered again, so be aware of the possibility of looping back to the same recovery routine. That recovery routine remains activated and can be entered again unless the recovery routine issued SETRP with REMREC=YES. If the recovery routine specified REMREC=YES, the system deactivated that recovery routine before giving control to the retry routine.
- Any previous recovery routines (those that percolated to the recovery routine that requested the retry) are deactivated.

Notes:

1. You can have as many retry points in your program as needed, and you can change the designated retry point as your mainline processing continues.
2. The retry routine can be a separate routine. The only requirement is that it must be in virtual storage. You are responsible for ensuring that the retry routine is in virtual storage when needed.

Recovery Routines that Percolate: When a recovery routine decides to percolate (or takes the default), it should do the following:

- Release resources that were acquired by the mainline, such as ENQs.
- Repair the cause of the error, if possible.
- Indicate the percolate option to the system. If an SDWA is present, the recovery routine issues the SETRP macro with the RC=0 parameter to indicate percolation. If no SDWA is present, the recovery routine has to set a return code of 0 in register 15.

Notes:

1. Once a recovery routine percolates, it is **no longer activated**; it cannot receive control again for this error.
2. An ESTAI routine can request that the system not give control to any further ESTAI routines by specifying RC=16 on the SETRP macro. The system then abnormally ends the task.

Understanding the Means of Communication

An important aspect of writing a recovery routine is understanding how the recovery routine communicates with the mainline routine, the retry routine, and the system. This section discusses the following means of communication:

- **Parameter area**

  The parameter area is set up by the mainline routine and passed to the recovery routine. See “Setting Up, Passing, and Accessing the Parameter Area.”

- **SDWA**

  The SDWA provides information to the recovery routine, and the recovery routine can communicate with the system, and with subsequent recovery routines, by placing information into the SDWA. See “Using the SDWA” on page 8-21.

- **Registers**

  When a recovery routine gets control, GPR 0 indicates whether an SDWA is available. When an SDWA is not available, the recovery routine can communicate its recovery options to the system only through registers. Aside from this circumstance, the recovery routine **cannot** use registers to communicate with the system; the routine must use the SDWA. Also, the mainline routine should not place information in registers and expect that information to be in the registers when the recovery routine gets control. Complete details about registers are in the section entitled “Understanding the Recovery Environment” on page 8-30.

You should understand that communications are handled differently depending on the following circumstances:

- Whether your recovery routine received an SDWA
- Whether the communication is with the recovery routine or with the retry routine.

Setting Up, Passing, and Accessing the Parameter Area

The primary means of communication between the mainline routine and the recovery routine is the parameter area that the mainline sets up and passes to the recovery routine. This section discusses:

- What your mainline routine should put into the parameter area
- How your mainline passes the parameter area to the recovery routine
- How your recovery routine accesses the parameter area.

Deciding What to Include in the Parameter Area: Your mainline routine can put whatever information it wants in the parameter area. Remember that the object is to provide the recovery routine with as much useful information as possible so the

recovery routine can be effective. Here are some suggestions for important information to place in the parameter area:

- The base registers for the mainline. The recovery routine must be able to establish addressability to whatever resources the mainline is holding.
- The addresses of all dynamically acquired storage.
- The location of a workarea for use by the recovery routine.
- Indications of what resources are held or serialized, such as ENQs, data sets, and so on.
- Footprints indicating the processing being performed by the mainline when the error occurred. Using footprints is a technique whereby the mainline sets bits as it goes through its processing. When the recovery routine gets control, it can check the parameter area to see which bits have been turned on, and thus can tell how far along the mainline was. The recovery routine can pinpoint what the mainline was doing at the time of error. If the mainline was done with its processing when the error occurred, the recovery routine might not need to retry, but might just clean up resources.
- An indication of whether a retry is desired.
- The input parameter list to the mainline. When the mainline received control, it might have received an input parameter list. The mainline can preserve this in the parameter area intended for use by the recovery routine. The recovery routine can then inspect the input parameter list to determine if the mainline received input that was not valid.
- Whatever register contents (both GPRs and ARs) the mainline wants to save (they might need to be restored upon retry).
- The location of important data areas used by the mainline. Errors often occur because of damage to information in a data area. The recovery routine might need to repair one or more of these data areas, and so must be able to access them. The recovery routine might also want to include these data areas when it specifies the areas of storage to dump.
- The addresses of any user-written routines available to repair damage. You might have separate routines designed to scan and repair queues, repair data areas, and so on. The recovery routine might want to call these other routines for assistance.

Passing the Parameter Area: When you provide a recovery routine, you have the opportunity to identify to the system the parameter area you want passed to the recovery routine. Here are the ways to accomplish that:

- **ESTAE and ESTAEX routines**

  Use the PARAM parameter on the ESTAE or ESTAEX macro to specify the address of the parameter area you have constructed.

- **ESTAI routines**

  Use the ESTAI parameter on the ATTACHX macro to specify both the address of the recovery routine to get control, and the address of the parameter area you have constructed.

Accessing the Parameter Area: Once the recovery routine gets control, the routine must know how to access the parameter area. That varies according to whether the system provided an SDWA, and according to how the recovery routine was defined:

- **SDWA is present**
  - **ESTAE macro**

    SDWAPARM and GPR 2 contain the address of the parameter area you specified on the PARAM parameter on ESTAE.
  - **ESTAEX macro**

    SDWAPARM contains the address of an 8-byte field, which contains the address and ALET of the parameter area you specified on the PARAM parameter on ESTAEX, and GPR 2 contains the address of the parameter area you specified on the PARAM parameter on ESTAEX. AR 2 contains the ALET qualifying the address in GPR 2.
  - **ATTACHX macro with ESTAI parameter**

    SDWAPARM and GPR 2 contain the address of the parameter area you specified on the ESTAI parameter on ATTACHX. The parameter area specified on ATTACHX is always assumed to be in the primary address space, so for AR-mode callers, the ALET is always zero.
- **SDWA is not present**
  - **ESTAE macro**

    GPR 2 contains the address of the parameter area you specified on the PARAM parameter on ESTAE.
  - **ESTAEX macro**

    GPR 2 contains the address of the parameter area you specified on the PARAM parameter on ESTAEX. AR 2 contains the ALET qualifying the address in GPR 2.
  - **ATTACHX macro with ESTAI parameter**

    GPR 2 contains the address of the parameter area you specified on the ESTAI parameter on ATTACHX. The parameter area specified on ATTACHX is always assumed to be in the primary address space, so for AR-mode callers, the ALET is always zero.

Using the SDWA

The SDWA is both a means by which the recovery routine can provide information to the system and to subsequent recovery routines, and a provider of information to the recovery routine. To access and update the SDWA, the recovery routine must include the IHASDWA mapping macro as a DSECT. For complete information about the SDWA, see SDWA in *OS/390 MVS Data Areas, Vol 4 (RD-SRRA)*. **The SDWA is always in the primary address space.**

Updating the SDWA: A recovery routine can update the SDWA in various ways:

- By issuing the SETRP macro (See “Using the SETRP Macro to Update the SDWA” on page 8-22.)

- By issuing the VRADATA macro (See the VRADATA macro in *OS/390 MVS Programming: Assembler Services Reference* and use of the VRADATA macro in “Symptoms Provided by a Recovery Routine” on page 9-6.)
- By directly updating specific fields (see “Important Fields in the SDWA” on page 8-23).

Using the SETRP Macro to Update the SDWA: Recovery routines issue the SETRP macro to communicate recovery options to the system, and to save serviceability data. The routine must have an SDWA to issue SETRP. The following are some of the things a recovery routine can do using the SETRP macro:

- **Indicate retry or percolate**

  Use the RC parameter on SETRP to let the system know whether the recovery routine wants to percolate (RC=0) or retry (RC=4). If attempting a retry, the routine must also specify a retry address on the RETADDR parameter.

  For ESTAI routines, you can also specify RC=16 to ask the system not to give control to any further ESTAI routines.

- **Specify register contents for the retry routine and free the SDWA**

  ESTAE-type recovery routines can use parameters on the SETRP macro to restore registers from the SDWA (RETREGS=YES), and to free the SDWA before control is given to the retry routine (FRESDWA=YES). See “Register Contents on Entry to a Retry Routine” on page 8-35 for information about using the RETREGS and FRESDWA parameters.

- **Save serviceability data**

  Use the RECPARM parameter to supply the load module name, the active CSECT name, and the recovery routine CSECT name.

- **Change the completion and reason codes**

  You can specify both completion and reason code values on the ABEND macro. The system passes these values to recovery routines in the SDWA. Recovery routines can change the values of the completion code and the reason code by using the SETRP macro. The COMPCOD parameter allows you to specify a new completion code; the REASON parameter allows you to specify a new reason code.

  The reason code has no meaning by itself, but must be used together with a completion code. To maintain meaningful completion and reason codes, the system propagates changes to these values according to the following rules:

  – If a user changes both the completion code and the reason code, the system accepts both new values.

  – If a user changes the reason code but not the completion code, the system accepts the new reason code and uses the unchanged completion code.

  – If a user changes the completion code but not the reason code, the system accepts the new completion code and uses a zero for the reason code.

Symptom Data Required in the SDWA for Dump Suppression: If the installation is using DAE to suppress duplicate dumps, the recovery routine must provide the following minimum data to enable dump suppression. See “Suppressing Dumps That Duplicate Previous Dumps” on page 9-2 for more information about dump suppression.

SDWA Field	Data	Example
SDWAMODN	Failing module name	IEAVTCXX
SDWACSCT	Failing CSECT name	IEAVTC22
SDWACID	Product or component identifier	SCDMP
SDWACIB		
SDWACIDB	Component identifier base	5655
SDWAREXN	Recovery routine name	IEAVTC2R
SDWASC	Subcomponent or module subfunction	RSM-PGFIX

Important Fields in the SDWA: The following table summarizes some of the key fields in the SDWA. Note that certain fields are in an extension of the SDWA called SDWARC1, which is a different DSECT. Here is how to access SDWARC1:

- SDWAXPAD in the SDWA contains the address of SDWAPTRS.
- SDWAPTRS is a DSECT which contains SDWASRVP.
- SDWASRVP contains the address of SDWARC1.

The fields described below that are in SDWARC1 are:

- SDWACRC
- SDWAARER
- SDWAARSV
- SDWACID
- SDWASC
- SDWAMLVL
- SDWARRL

Figure 8-5 (Page 1 of 4). Key Fields in the SDWA

Field Name	Use
SDWAPARM	This 4-byte field contains the address of the user parameter area that you supply for an ESTAE-type recovery routine. For routines defined by the ESTAEX macro, this field contains the address of an 8-byte area. The first four bytes of this 8-byte area contain the address of the parameter area you specified on the ESTAEX macro; the next four bytes contain the ALET for the parameter area.
SDWACMPC	This 3-byte field contains the completion code that existed when the system gave control to the recovery routine. The recovery routine can change the completion code by issuing the SETRP macro with the COMPCOD parameter. The system completion code appears in the first twelve bits, and the user completion code appears in the second twelve bits.
SDWARPIV	This bit tells the recovery routine that the registers and PSW at the time of error are not available. When this bit is on, the contents of SDWAGRSV, SDWAARER, and SDWAEC1 are unpredictable.

Figure 8-5 (Page 2 of 4). Key Fields in the SDWA

Field Name	Use
SDWACRC	This 4-byte field contains the reason code associated with the completion code in SDWACMPC. The reason code is set through the REASON parameter of the ABEND macro, and is valid only when bit SDWARCF is on. The recovery routine may change this reason code by specifying a new value for the REASON parameter of the SETRP macro. **Note:** This reason code is **not** the same as the return code that programs may set in GPR 15 before they issue the ABEND macro.
SDWARCF	If on, this bit indicates that SDWACRC contains a reason code.
SDWAGRSV	This field contains the contents of the general purpose registers (GPRs) 0-15 as they were at the time of the error.
SDWAARER	This field contains the contents of the access registers (ARs) 0-15 as they were at the time of the error.
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 ESTAE-type recovery routines (except for ESTAI routines): If a program establishes an ESTAE routine, and subsequently performs a stacking operation while running under the same RB as when it established the ESTAE routine, SDWAEC2 contains the PSW from the linkage stack entry immediately following the entry that was current when the ESTAE routine was established. Otherwise, SDWAEC2 contains the current RBOPSW from the RB that activated the recovery routine, and the PSW is the one from the time of the last interruption of that RB that occurred while the RB was unlocked and enabled. Bit SDWAINTF in SDWAXFLG indicates whether the contents of SDWAEC2 are from the linkage stack (SDWAINTF is 1) or from an RB (SDWAINTF is 0). • For an ESTAI routine, this field contains zero.

Figure 8-5 (Page 3 of 4). Key Fields in the SDWA

Field Name	Use
SDWASRSV	The contents of this field vary according to the type of recovery routine: • For ESTAE-type recovery routines (except for ESTAI routines): If a program establishes an ESTAE routine, and subsequently performs a stacking operation while running under the same RB as when it established the ESTAE routine, SDWASRSV contains GPRs 0-15 from the linkage stack entry immediately following the entry that was current when the ESTAE routine was established. Otherwise, SDWASRSV contains GPRs 0-15 from the RB that activated the recovery routine, and the GPRs are the same as they were at the time of the last interruption of that RB that occurred while the RB was unlocked and enabled. Bit SDWAINTF in SDWAXFLG indicates whether the contents of SDWASRSV are from the linkage stack (SDWAINTF is 1) or from an RB (SDWAINTF is 0). • For an ESTAI routine, this field contains zeros. If the recovery routine requests a retry, the system might use the contents of this field to load the GPRs for the retry routine. See the RETREGS parameter description in the SETRP macro in *OS/390 MVS Programming: Assembler Services Reference* for details. To change the contents of the GPRs for the retry routine, you must make the changes to SDWASRSV and then issue SETRP with RETREGS=YES. You can update the registers directly or with the RUB parameter on SETRP.
SDWAARSV	The contents of this field depend on the type of recovery routine: • For ESTAE-type recovery routines (except for ESTAI routines): If a program establishes an ESTAE routine, and subsequently performs a stacking operation while running under the same RB as when it established the ESTAE routine, SDWAARSV contains ARs 0-15 from the linkage stack entry immediately following the entry that was current when the ESTAE routine was established. Otherwise, SDWAARSV contains ARs 0-15 from the RB that activated the recovery routine, and the ARs are the same as they were at the time of the last interruption of that RB that occurred while the RB was unlocked and enabled. Bit SDWAINTF in SDWAXFLG indicates whether the contents of SDWAARSV are from the linkage stack (SDWAINTF is 1) or from an RB (SDWAINTF is 0). • For an ESTAI routine, this field contains zeros. If the recovery routine requests a retry, the system might use the contents of this field to load the ARs for the retry routine. See the RETREGS parameter description in the SETRP macro in *OS/390 MVS Programming: Assembler Services Reference* for details. To change the contents of the ARs for the retry routine, you must make the changes in SDWAARSV, and then issue SETRP with RETREGS=YES.
SDWASPID	This field contains the subpool ID of the storage used to obtain the SDWA, for use whenever the retry routine is responsible for freeing the SDWA.

Figure 8-5 (Page 4 of 4). Key Fields in the SDWA	
Field Name	**Use**
SDWALNTH	This field contains the length, in bytes, of this SDWA, the SDWA extensions, and the variable recording area, for use whenever the retry routine is responsible for freeing the SDWA. (This allows the retry routine to free the extensions along with the SDWA.)
SDWACOMU	The recovery routines can use this 8-byte field to communicate with each other when percolation occurs. The system copies this field from one SDWA to the next on all percolations. When the field contains all zeros, either no information is passed or the system has not been able to pass the information.
SDWATRAN	This field contains one of the following if a translation exception occurred: • The valid translation exception address if the SDWATEAV bit is 1. • The ASID of the address space in which the translation exception occurred if the SDWATEIV bit is 1. If both the SDWATEAV and SDWATEIV bits are 0, ignore the SDWATRAN field.
SDWATEAR	For translation exceptions that occur in AR mode, this 1-byte field identifies the number of the AR that the program was using when the translation exception occurred.
SDWACLUP	If on, this bit indicates that the recovery routine cannot retry.
SDWAPERC	If on, this bit indicates that a recovery routine has already percolated for this error.
SDWAEAS	If on, this bit indicates that a previous recovery routine provided sufficient diagnostic information pertaining to this error. The recovery routine providing the information is responsible for setting the bit.
SDWACID	The recovery routine can use this 5-byte field to provide the component ID of the component involved in the error.
SDWASC	The recovery routine can use this 23-byte field to provide the name of the component and a description of the function or subfunction involved in the error.
SDWAMLVL	The recovery routine can use this 16-byte field to indicate the level of the module involved in the error. The first 8 bytes contains the date (SDWAMDAT) and the second 8 bytes contains the version (SDWAMVRS).
SDWARRL	The recovery routine can use this 8-byte field to indicate the recovery routine's entry point label.
SDWALSLV	The recovery routine can use this 2-byte field to control the linkage stack state upon retry. See “Linkage Stack at Time of Retry” on page 8-41 for additional information.

Special Considerations for ESTAE-Type Recovery Routines

This section discusses some special considerations for writing ESTAE-type recovery routines:

- RB considerations
- Linkage stack considerations
- Outstanding I/Os at time of failure
- Other considerations for ESTAE-type recovery routines

RB Considerations

A program must activate and deactivate ESTAE-type recovery routines under the same RB level. If you try to deactivate an ESTAE-type recovery routine that is not associated with your RB, you get a return code that indicates your request is not valid.

ESTAE-type recovery routines are deactivated when their associated RBs terminate. This is important because a program expects one of its own ESTAE-type recovery routines to get control rather than one left behind by a called program. A program might, however, invoke a service routine that does not create an RB. If that routine then issues an ESTAEX or ESTAE macro and fails to deactivate the resulting ESTAE-type recovery routine, a problem could develop if the original program encounters an error. The ESTAE-type recovery routine left behind by the service routine would receive control rather than the ESTAE-type recovery routine associated with the program, because the recovery routine specified by the most recently issued ESTAE or ESTAEX macro gets control.

IBM recommends that every program that activates an ESTAE-type recovery routine also deactivate it.

For retry from an ESTAE-type recovery routine, the retry routine runs as a continuation of the code that activated the recovery routine. That is, the retry routine runs under the same RB that defined the ESTAE-type recovery routine, and the system purges all RBs created after the retry RB before giving control to the retry routine.

Note that ESTAI is an exception; a retry request from a recovery routine defined by the ESTAI parameter of the ATTACHX macro must run under a program request block (PRB). The retry routine cannot run under the PRB of the routine that defined the ESTAI routine, because that PRB is associated with a different task. The system scans the RB queue associated with the task under which the retry is to occur, starting with the RB that was interrupted (the newest RB). The system then uses the following rules to select a PRB for the retry routine:

- If one or more PRBs exist that represent an ESTAE-type recovery routine, use the newest one.
- If no PRBs exist that represent ESTAE-type recovery routines, use the newest PRB that does not have any non-PRBs (such as SVRBs) that are older.

If the RB queue contains no PRBs at all, retry is suppressed.

Linkage Stack Considerations

Consider the following information about the linkage stack when writing an ESTAE-type recovery routine or a retry routine, or when deactivating an ESTAE-type recovery routine:

Recovery Routine: **IBM recommends** that your recovery routine not modify or extract from the linkage stack entry that is current when the routine is entered. In some cases, the system might prevent an ESTAE-type recovery routine from modifying or extracting from that linkage stack entry. If your recovery routine attempts to modify or extract from the linkage stack entry when the system does not allow it, the result is a linkage stack exception.

IBM recommends that if your recovery routine adds entries to the linkage stack, through a stacking PC or BAKR instruction, it should also remove them. If the recovery routine adds entries to the stack and does not remove them, the system recognizes an error when the recovery routine returns control. If the recovery routine retries, the additional entries **are not** given to the retry routine. If the recovery routine percolates, subsequent recovery routines receive a linkage stack with entries more recent than the entry that was current at the time of error.

Retry Routine: When the system gives control to your retry routine, the linkage stack level is set to the level that was current when your program activated the recovery routine, unless the recovery routine sets the SDWALSLV field.

Deactivating an ESTAE-Type Recovery Routine: A program may deactivate an ESTAE-type recovery routine only under the same linkage stack level as the level that existed when the program activated the recovery routine. This rule affects programs that add entries to the linkage stack either through the BAKR or PC instruction. Failure to follow this rule results in an error return code of 36 from the ESTAE or ESTAEX macro.

When you issue a PR, the system automatically deactivates all ESTAE-type recovery routines that were previously activated under that current linkage stack entry.

Outstanding I/Os at the Time of Failure

Before the most recently activated ESTAE-type recovery routine receives control, the system can handle outstanding I/Os at the time of the failure. You request this through the macro that defines the routine (that is, through the PURGE parameter on ESTAE, ESTAEX, or ATTACHX). The system performs the requested I/O processing only for the first ESTAE-type recovery routine that gets control. Subsequent routines that get control receive an indication of the I/O processing previously done, but no additional processing is performed.

Note: You should understand PURGE processing before using this parameter. For information on where PURGE processing is documented, see *DFSMS/MVS General Information* for the version of DFP you have installed.

If there are quiesced restorable I/O operations (because you specified PURGE=QUIESCE on the macro for the most recently defined ESTAE-type recovery routine), the retry routine can restore them as follows:

- If the recovery routine specified FRESDWA=YES and RETREGS=NO on the SETRP macro, or the system did not provide an SDWA, the system supplies the address of the purged I/O restore list in GPR 2 on entry to the retry routine.
- If the recovery routine specified FRESDWA=NO and RETREGS=NO on the SETRP macro, GPR 1 contains the address of the SDWA, and the address of the purged I/O restore list is in the SDWAFIOB field on entry to the retry routine.
- If the recovery routine specified FRESDWA=NO and RETREGS=YES on the SETRP macro, the recovery routine must pass the address of the SDWA to the retry routine (in the user parameter area, or in GPR 0). The address of the purged I/O restore list is in the SDWAFIOB field on entry to the retry routine.
- If the recovery routine specified FRESDWA=YES and RETREGS=YES on the SETRP macro, the retry routine cannot access the purged I/O restore list,

unless the recovery routine saves the address of the purged I/O restore list from SDWAFIOB, and passes the address to the retry routine.

The following table provides a summary of how the retry routine can access quiesced restorable I/O operations:

Figure 8-6. Restoring Quiesced Restorable I/O Operations

Parameter on SETRP Macro	RETREGS=NO	RETREGS=YES
FRESDWA=YES	GPR 2 contains the address of the purged I/O restore list (see note below)	Retry routine cannot access the purged I/O restore list, unless the recovery routine saves the address from SDWAFIOB and passes the address to the retry routine
FRESDWA=NO	GPR 1 contains the address of the SDWA; SDWAFIOB contains the address of the purged I/O restore list	The recovery routine must pass the address of the SDWA to the retry routine; SDWAFIOB contains the address of the purged I/O restore list.
Note: If the system did not provide an SDWA and RETREGS=NO, then GPR 2 contains the address of the purged I/O restore list.		

You can use the RESTORE macro to have the system restore all I/O requests on the list. For information about where the RESTORE macro is documented, see *DFSMS/MVS General Information* for the version of DFP you have installed.

Additional Considerations Specific to ESTAE-Type Recovery Routines

The following are additional things you should consider that are specific to ESTAE-type recovery routines:

- During processing of the first and all subsequent recovery routines, the system allows or disallows asynchronous processing (such as a timer exit) depending on how you specify the ASYNCH parameter when you define the routine (that is, through the ASYNCH parameter on ESTAE, ESTAEX, and ATTACHX).
- The following list describes what the system does when it is done processing a particular recovery routine (either because the recovery routine percolates, or because the recovery routine itself encounters an error and has no recovery routine of its own that retries):
  - Accumulates dump options
  - Resets the asynchronous exit indicator according to the request of the next recovery routine
  - Ignores the I/O options for the next recovery routine
  - Initializes a new SDWA
  - Gives control to the next recovery routine.

  If all recovery routines fail or percolate, the task is terminated.
- If a non-job step task issues an ABEND macro with the STEP parameter, the system gives control to recovery routines for the non-job step task. If the recovery routines do not request a retry, the job step is terminated with the specified completion code. Subsequent recovery routines for the job step task get control only when you specify TERM=YES on the macros that defined

those recovery routines. You can specify TERM=YES on ESTAE, ESTAEX, and ATTACHX.

If the recovery routines for the job step task do not retry, subsequent recovery routines for any other non-job step tasks get control in the same way they would if the job step task itself encountered the error and then did not retry.

- For some situations, the system gives control to ESTAE-type recovery routines **only** when the TERM=YES parameter was specified. The situations are:
  - System-initiated logoff
  - Job step timer expiration
  - Wait time limit for job step exceeded
  - DETACH macro was issued from a higher level task (possibly by the system if the higher level task encountered an error)
  - Operator cancel
  - Error occurred on a higher level task
  - Error in the job step task when a non-job step task issued the ABEND macro with the STEP parameter
  - OpenMVS is canceled and the user's task is in a wait in the OpenMVS kernel.

  When the system gives control to the recovery routines defined with the TERM=YES parameter as a result of the above errors, the system takes the following actions:
  - Sets the SDWACLUP bit
  - Gives control to all such routines in LIFO order
  - Does not enter any ESTAI routine previously suppressed by a return code of 16, or any previously entered recovery routine that requested percolation
  - Ignores any request for retry.

Understanding the Recovery Environment

When you write a recovery routine, you must take into consideration a number of environmental factors that are present when the recovery routine gets control, and that are present when a retry routine gets control. This section discusses environmental factors in two broad categories, distinguishing register contents from all other environmental factors:

- **Register contents.**

  Recovery routines are interested in register contents at the following times:

  - When the error occurs

    When the recovery routine gets control, certain fields in the SDWA contain the register contents at the time the error occurs. SDWAGRSV contains the contents of the GPRs; SDWAARER contains the contents of the ARs.

  - On entry to and return from the recovery routine

    See “Register Contents on Entry to a Recovery Routine” on page 8-32 and “Register Contents on Return from a Recovery Routine” on page 8-34 for details.

  - On entry to the retry routine

See “Register Contents on Entry to a Retry Routine” on page 8-35 for details.

- **All other environmental factors.**

  The other environmental factors important in a recovery environment are:

  – Authorization: problem state or supervisor state, PSW key, and PSW key mask (PKM)

  – SDWA storage key

  – Dispatchable unit mode

  – AMODE

  – ASC mode

  – Interrupt status

  – DU-AL

  – Program mask

  – Condition of the linkage stack

This section discusses each of the environmental factors, and makes distinctions, where necessary, that depend on the following:

- Whether the system provided an SDWA
- Whether you are dealing with the recovery routine or the retry routine.

Register Contents

This section describes register contents for the following:

- On entry to a recovery routine
- On return from a recovery routine (see “Register Contents on Return from a Recovery Routine” on page 8-34)
- On entry to a retry routine.

The following table provides a roadmap to all the tables containing register content information **on entry to a recovery routine** or **on entry to a retry routine**:

Figure 8-7. Where to Find Register Content Information	
Registers Described For:	**Table and Page Number:**
ESTAE-type recovery routine with an SDWA	Figure 8-8 on page 8-32
ESTAE-type recovery routine without an SDWA	Figure 8-9 on page 8-34
Retry from an ESTAE-type recovery routine without an SDWA	Figure 8-10 on page 8-35
Retry from an ESTAE-type recovery routine with an SDWA, RETREGS=NO, and FRESDWA=NO	Figure 8-11 on page 8-35
Retry from an ESTAE-type recovery routine with an SDWA, RETREGS=NO, and FRESDWA=YES	Figure 8-12 on page 8-36
Retry from an ESTAE-type recovery routine with an SDWA and RETREGS=YES	Figure 8-13 on page 8-37

Register Contents on Entry to a Recovery Routine

The register contents on entry to an ESTAE-type recovery routine are different depending on whether the system supplied and SDWA. The following tables describe the register contents on entry to the recovery routine for both situations.

Figure 8-8. Register Contents—ESTAE-Type Recovery Routine With an SDWA

Register	Contents
General Purpose Registers	
GPR 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 (X'10')** No I/O processing was performed.
GPR 1	Address of the SDWA.
GPR 2	One of the following: • If you specified the PARAM parameter on ESTAE, ESTAEX, or ATTACHX, the address of the user-supplied parameter area • If you issued ESTAE, ESTAEX, or ATTACHX without the PARAM parameter, zero.
GPRs 3 - 12	Do not contain any information for use by the routine.
GPR 13	Address of a 72-byte register save area.
GPR 14	Return address to the system.
GPR 15	Entry point address of the ESTAE-type recovery routine.
Access Registers	
ARs 0 - 1	Zero
AR 2	One of the following: • If you issued the ESTAEX macro in AR ASC mode, an ALET that qualifies the address in GPR 2. • Otherwise, this register does not contain any information for use by the routine.
ARs 3 - 15	Zero.

Figure 8-9. Register Contents—ESTAE-Type Recovery Routine Without an SDWA	
Register	**Contents**
General Purpose Registers	
GPR 0	12 (X'0C'). The system could not obtain an SDWA.
GPR 1	Completion code in bytes 1-3. The system completion code appears in the first 12 bits, and the user completion code appears in the second 12 bits.
GPR 2	One of the following: • If you specified the PARAM parameter on ESTAE, ESTAEX, or ATTACHX, the address of the user-supplied parameter area • If you issued ESTAE, ESTAEX, or ATTACHX without the PARAM parameter, zero.
GPRs 3 - 13	Do not contain any information for use by the routine. **Note:** When the system does not provide an SDWA, GPR 13 **does not** contain the address of a standard 72-byte save area. In this case, your ESTAE-type recovery routine must save the address from GPR 14 and use it as the return address to the system.
GPR 14	Return address to the system.
GPR 15	Entry point address of the ESTAE-type recovery routine.
Access Registers	
ARs 0 - 1	Zero
AR 2	One of the following: • If you issued the ESTAEX macro in AR ASC mode, an ALET that qualifies the address in GPR 2. • Otherwise, this register does not contain any information for use by the routine.
ARs 3 - 15	Zero.

Register Contents on Return from a Recovery Routine

The register contents on return from a recovery routine depend on whether the system provided an SDWA. ESTAE-type recovery routines that receive an SDWA can use any register without saving its contents, except GPR 14. The routines must maintain the return address supplied in GPR 14. The routines do not have to place any information in the registers for use by the system.

ESTAE-type recovery routines that **do not** receive an SDWA must set one of the following return codes in GPR 15:

Return Code	Meaning
0	The recovery routine requests percolation.
4	The recovery routine requests a retry. **The recovery routine must then place the address of the retry routine in GPR 0.**
16 (X'10')	Valid only for an ESTAI recovery routine. The system should not give control to any further ESTAI routines, and should abnormally end the task.

Register Contents on Entry to a Retry Routine

The register contents on entry to a retry routine vary according to the following:

- Whether an SDWA is present.
- If an SDWA is present, what the recovery routine specifies on the SETRP macro.

The parameters on SETRP that affect register contents on entry to the retry routine from an ESTAE-type recovery routine are the following:

- The RETREGS parameter controls whether registers are restored from the SDWA. If you specify RETREGS=NO, registers are not restored from the SDWA.

  If you specify RETREGS=YES, GPRs are restored from SDWASRSV, and ARs are restored from SDWAARSV. If you specify RETREGS=YES,RUB, you can manipulate the contents of SDWASRSV to whatever you wish the GPRs to be before they are restored. Or, you can directly manipulate the contents of both SDWASRSV and SDWAARSV.

  See the description of the SETRP macro in *OS/390 MVS Programming: Assembler Services Reference* for complete details.

- The FRESDWA parameter controls whether the system frees the SDWA before giving control to the retry routine. FRESDWA=YES instructs the system to free the SDWA; FRESDWA=NO instructs the system not to free the SDWA. This has an affect on the register contents on entry to the retry routine.

The following tables describe the register contents under various circumstances on entry to a retry routine from an ESTAE-type recovery routine:

Figure 8-10. Register Contents—Retry from an ESTAE-Type Recovery Routine Without an SDWA

Register	Contents
General Purpose Registers	
GPR 0	12 (X'0C').
GPR 1	Address of the user parameter area.
GPR 2	Address of the purged I/O restore list if I/O was quiesced and is restorable; otherwise, zero.
GPRs 3 - 14	Do not contain any information for use by the routine.
GPR 15	Entry point address of the retry routine.
Access Registers	
AR 0	Zero.
AR 1	One of the following: • If the recovery routine was defined using ESTAEX and the caller was in AR ASC mode, the ALET of the user parameter area. • Otherwise, this register does not contain any information for use by the routine.
ARs 2 - 13	Do not contain any information for use by the routine.
ARs 14 - 15	Zero.

Figure 8-11. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=NO

Register	Contents
General Purpose Registers	
GPR 0	Zero.
GPR 1	Address of the SDWA.
GPRs 2 - 14	Do not contain any information for use by the routine.
GPR 15	Entry point address of the retry routine.
Access Registers	
ARs 0 - 1	Zero.
ARs 2 - 13	Do not contain any information for use by the routine.
ARs 14 - 15	Zero.

Figure 8-12. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=YES

Register	Contents
General Purpose Registers	
GPR 0	20 (X'14').
GPR 1	Address of the user parameter area.
GPR 2	Address of the purged I/O restore list, if I/O was quiesced and is restorable; otherwise, zero.
GPRs 3 - 14	Do not contain any information for use by the routine.
GPR 15	Entry point address of the retry routine.
Access Registers	
AR 0	Zero.
AR 1	One of the following: • If the recovery routine was defined using ESTAEX and the caller was in AR ASC mode, the ALET of the user parameter area. • Otherwise, this register does not contain any information for use by the routine.
ARs 2 - 13	Do not contain any information for use by the routine.
ARs 14 - 15	Zero.

Figure 8-13. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA and RETREGS=YES	
Register	**Contents**
General Purpose Registers	
GPRs 0 - 15	Restored from SDWASRSV, regardless of whether the recovery routine specified FRESDWA=NO or FRESDWA=YES. Note that register 15 does not contain the entry point address of the retry routine unless the recovery routine sets it up that way.
Access Registers	
ARs 0 - 15	Restored from SDWAARSV, regardless of whether the recovery routine specified FRESDWA=NO or FRESDWA=YES.

Other Environmental Factors in Recovery

As mentioned previously, the other environmental factors to be concerned about in a recovery environment are:

- Authorization: problem state or supervisor state, PSW key, and PKM
- SDWA storage key
- Dispatchable unit mode
- AMODE
- ASC mode
- Interrupt status
- DU-AL
- Program mask
- Condition of the linkage stack

These environmental factors differ depending on whether you are dealing with the recovery routine or the retry routine.

Environment on Entry to an ESTAE-Type Recovery Routine

The following is a description of each environmental factor on entry to an ESTAE-type recovery routine.

Authorization:

- **Problem or supervisor state**

  The ESTAE-type recovery routines you can write are entered in problem state.

- **PSW key**

  An ESTAE-type recovery routine is entered with the PSW key that existed at the time the recovery routine was defined.

- **PKM**

  An ESTAE-type recovery routine is entered with the PKM that existed at the time the recovery routine was defined.

SDWA Storage Key: An ESTAE-type recovery routine receives an SDWA in the same storage key as the TCB key at the time the related task made the first storage request from subpool 0.

Dispatchable Unit Mode: All ESTAE-type recovery routines receive control in task mode.

AMODE: A recovery routine defined through the ESTAE macro, or the ESTAI parameter on ATTACHX, has the addressing mode of the caller at the time the macro was issued. ESTAEX routines are always entered in 31-bit addressing mode.

ASC Mode: A recovery routine defined through the ESTAE macro is entered in primary ASC mode. A recovery routine defined through the ESTAEX macro or the ESTAI parameter on ATTACHX is entered in the ASC mode that existed at the time the macro was issued.

Interrupt Status: All ESTAE-type recovery routines are entered enabled for I/O and external interrupts.

DU-AL: ESTAE-type recovery routines receive control with the DU-AL that was current at the time of the error, as modified by any previous recovery routines, with the following exception. For an ESTAE-type recovery routine activated by an IRB, or activated by an IRB's ESTAE-type recovery routine, the ESTAE-type recovery routine receives the IRB's DU-AL (IRBs get control with their own DU-AL). The system does not modify the contents of the DU-AL during recovery processing.

Program Mask: The program mask on entry to an ESTAE-type recovery routine is the same as the program mask at the time of error.

Condition of the Linkage Stack: On entry to an ESTAE-type recovery routine, the current linkage stack entry is the same as it was at the time of the error, unless a previous recovery routine added entries to the linkage stack through a PC or BAKR instruction and did not remove them. In such a case, when percolation occurs and the recovery routine gets control, the linkage stack contains additional entries beyond what was the current entry at the time of the error for which the recovery routine received control. **IBM recommends** that any recovery routines that add entries to the linkage stack also remove them.

Restricted Environments: During **normal** task termination, a resource manager might end abnormally; its own recovery routines, if any exist, will receive control. If they do not retry, or if the resource manager has no recovery routines, the system now considers this situation to be an **abnormal** termination, and passes control to the newest ESTAI routine. Because the abending resource manager, and any previous resource managers, might have completed some processing, the ESTAI routine will run in an unpredictable environment. In this situation, IBM recommends that you restrict the ESTAI routine's processing. For the ESTAI routine to run in this environment, design it to:

1. Check the STCBRMET field in the STCB; if the bit is on, the ESTAI routine is running **after** a resource manager has ended abnormally and its recovery routines have not retried. In this situation, the ESTAI routine does not need to hold a lock to check the STCBRMET field. See *OS/390 MVS Data Areas, Vol 5 (SSAG-XTLST)* for the mapping of the STCB.
2. Do as little processing as possible, and nothing that might depend on a resource that might have been cleaned up already.
3. Do **not** request to retry. The system will not allow a retry in this situation.

Note that no other ESTAE-type routines receive control in this situation; only those established through the ATTACHX macro still exist at this point in termination processing.

Environment on Entry to a Retry Routine from an ESTAE-Type Recovery Routine

The following is a description of each environmental factor on entry to a retry routine that was specified by an ESTAE-type recovery routine.

Authorization:

- **Problem or supervisor state**

  The retry routine from an ESTAE-type recovery routine that you can write is entered in problem state.

- **PSW key**

  If the recovery routine was defined by an ESTAE or ESTAEX macro, the retry routine is entered with the same PSW key that existed when the macro was issued.

  If the recovery routine was defined by the ESTAI parameter of the ATTACHX macro, the retry routine is entered with the same PSW key as the one in RBOPSW of the retry RB, if the RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem state, and the PKM of that RB does not have authority to keys less than 8. Otherwise, the PSW key of the retry routine is that of the task in error.

- **PKM**

  If the recovery routine was defined through the ESTAE or ESTAEX macro, the retry routine is entered with the PKM that existed when the macro was issued.

  If the recovery routine was defined through the ESTAI parameter of the ATTACHX macro, the retry routine is entered with the PKM from the retry RB if the RBOPSW of the retry RB has a key greater that or equal to 8 and is in problem state, and the PKM of that RB does not have authority to keys less than 8. Otherwise, the PKM of the retry routine has authority that is equivalent to that of the task in error.

SDWA Storage Key: If the recovery routine does not request that the system free the SDWA, the retry routine receives the SDWA in the same storage key as that which the recovery routine received.

Dispatchable Unit Mode: The retry routine is always entered in task mode.

AMODE: Retry routines are entered in the same addressing mode that existed when the recovery routine was entered. Remember that ESTAEX routines are always entered in 31-bit addressing mode. The recovery routine cannot change the addressing mode of the retry routine.

ASC Mode: For recovery routines defined through the ESTAE macro, the retry routine is entered in primary ASC mode. For recovery routines defined through the ESTAEX macro or through the ESTAI parameter on ATTACHX, the retry routine is entered with the ASC mode of the caller when the macro was issued.

Interrupt Status: The retry routine is always entered enabled for I/O and external interrupts.

DU-AL: The retry routine is entered with the same DU-AL that the ESTAE-type recovery routine received, as modified by the ESTAE-type recovery routine. The system does not modify the contents of the DU-AL during recovery processing.

Program Mask: When the retry routine receives control, the program mask is the one in the RBOPSW for the retry RB, saved at the time of the last interruption of that RB that occurred while the RB was unlocked and enabled.

Condition of the Linkage Stack: For recovery routines defined through the ESTAE or ESTAEX macro, on entry to the retry routine, the current linkage stack entry is the same as it was at the time the macro was issued, unless the recovery routine has set the SDWALSLV field.

For recovery routines defined through the ESTAI parameter on ATTACHX, on entry to the retry routine, the current linkage stack entry is the same as it was at the time the selected retry RB was entered, unless the recovery routine has set the SDWALSLV field.

Summary of Environment on Entry to an ESTAE-Type Recovery Routine and Its Retry Routine

Figure 8-14 summarizes some of the environmental factors for ESTAE-type recovery routines under different conditions. Specifically, the table lists the status information of:

- The caller at the time of issuing the macro
- The recovery routine at entry
- The retry routine at entry.

Figure 8-14. Environments of ESTAE-type Recovery Routines and their Retry Routines

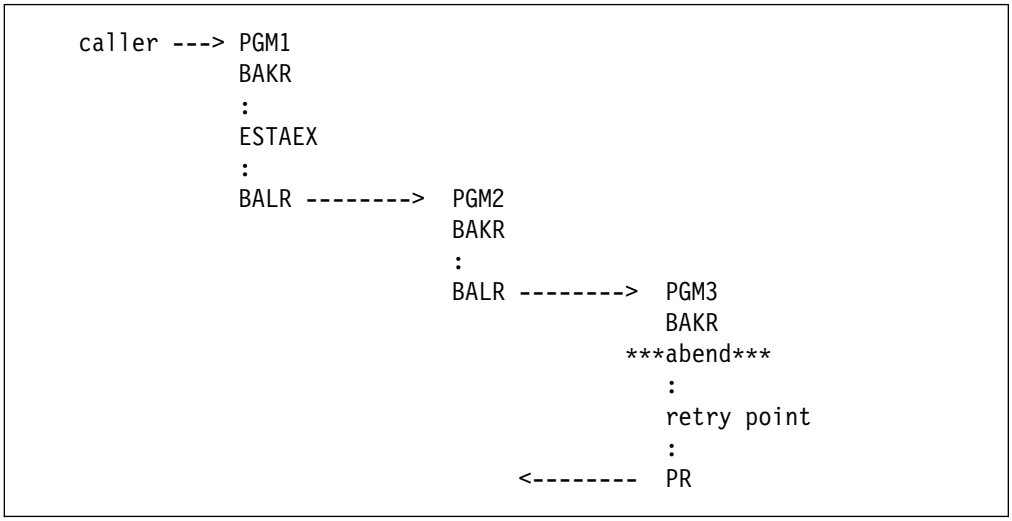
Type of Recovery	Environment		
	When macro was issued	**At entry to recovery routine**	**At entry to retry routine**
ESTAE	ASC mode=primary	ASC mode=primary Linkage stack at time of error (see Note 1) PKM at time macro was issued	ASC mode=primary Linkage stack at time macro was issued (see Note 2) PKM at time macro was issued
ESTAEX	ASC mode=primary or AR	ASC mode at time macro was issued Linkage stack at time of error (see Note 1) PKM at time macro was issued	ASC mode at time macro was issued Linkage stack at time macro was issued (see Note 2) PKM at time macro was issued
ESTAI (through ATTACHX)	ASC mode=primary or AR	ASC mode at time macro was issued Linkage stack at time of error (see Note 1) PKM at time macro was issued For possible environment restrictions, see “Restricted Environments” on page 8-38.	ASC mode at time macro was issued Linkage stack at time the retry RB was entered (see Note 2) For PKM, see the complete description on page 8-39

Notes:

1. The linkage stack presented to the recovery routine might have additional entries, beyond what was the current entry at the time of the error, if a previous recovery routine added entries through a PC or BAKR instruction and did not remove them.
2. The linkage stack presented to the retry routine might have additional entries, beyond what was current at the time that the routine was activated, when the recovery routine set the SDWALSLV field.
3. At time of entry to the recovery routine, the AMODE will be the same as the time of invocation, except for ESTAEX routines, which are always given control in AMODE 31.
4. The AMODE at the retry point will be the same as the AMODE on entry to the recovery routine.
5. An ESTAE-type recovery routine is entered with the PSW key that existed at the time it was defined.

Linkage Stack at Time of Retry

There is one retry situation you must avoid: the situation where the retry routine runs with a linkage stack entry that is inappropriate. Consider the following example, where PGM1 activates an ESTAEX routine that handles recovery for PGM1, PGM2, and PGM3.



Both PGM2 and PGM3 use the BAKR instruction to save status; each BAKR adds an entry to the linkage stack. Within PGM3, “retry point” indicates the location where the ESTAEX routine is to retry. After PGM3 issues the BAKR instruction, the last entries in the linkage stack are:

- Entry 1 -- caused by PGM1's BAKR
- Entry 2 -- caused by PGM2's BAKR
- Entry 3 -- caused by PGM3's BAKR

When the abend occurs in PGM3, unless you take special measures, the linkage stack level is reset to the level that was current when PGM1 activated the ESTAEX recovery routine. However, retry from the abend in PGM3 occurs within PGM3. The linkage stack level and the retry routine are not “in synch.” Measures you take to avoid this situation involve:

1. Passing the recovery routine a value that represents the difference between the level of the linkage stack that the retry routine in PGM3 needs and the level of the stack at the time PGM1 activated the ESTAEX routine. (In our example, the difference is 2 entries.)
2. Having the recovery routine set the value “2” in the SDWALSLV field in the SDWA.

At a retry, the system uses the value in SDWALSLV to adjust the linkage stack. In this way, the retry routine has the appropriate current linkage stack entry.

Two ways your program can track the entries in the linkage stack are:

- Count the number of entries added to the stack through BAKRs since PGM1 activated the ESTAEX routine. Subtract from that total the number of entries taken from the stack through corresponding PRs.
- Issue the IEALSQRY macro, which returns the number as output.

In either case, the recovery routine must receive the value and must place it in SDWALSLV. In summary, the value in SDWALSLV is the difference between the number of linkage stack entries present when the retry routine gets control and the number that were present when the recovery routine was activated. The system preserves the additional entries on the linkage stack for use by the retry routine. These linkage stack entries must exist at the time of the error; the system does not create any new entries.

The following rules apply to the value in SDWALSLV, as it pertains to linkage stack entries:

- The system ignores the SDWALSLV field when retry is from a STAE or STAI recovery routine.
- The value must not reflect entries that were placed on the linkage stack through a PC instruction.
- The value must reflect only those entries associated with programs that are problem state and running with the same PSW key as the program that activated the ESTAE-type recovery routine.
- For ESTAE-type routines, the value must reflect only those entries associated with programs that have been established by a program running under the RB of the retry routine. See “RB Considerations” on page 8-27.

If any of these rules are broken, retry still occurs but the system ignores the entry that did not conform and all subsequent entries.

Understanding Recovery through a Coded Example

This section provides a coded example illustrating a mainline routine with both a recovery routine and a retry routine as entry points in the mainline code.

The code in this example does not contain any real function. The mainline code does little more than save status, establish addressability, obtain a dynamic area (making the code reentrant), define a recovery routine, and issue the ABEND macro to pass control to the system.

The purpose of the example is just to illustrate how you might code a program that contains both a recovery routine and a retry routine, and how the three routines interact. The example also illustrates how you design an ESTAE-type recovery routine to allow for the possibility that the system might not provide an SDWA.

```
EXAMPLE   CSECT                 * SAMPLE PROGRAM THAT USES ESTAEX
EXAMPLE   AMODE 31
EXAMPLE   RMODE ANY
          USING EXAMPLE,15      * ESTABLISH TEMPORARY ADDRESSABILITY
          B    @PROLOG          * BRANCH AROUND EYE CATCHER
          DC   CL24'EXAMPLE 04/10/92.01' * EYE CATCHER
*
* USE THE LINKAGE STACK TO SAVE STATUS ON ENTRY TO THE PROGRAM.
*
@PROLOG   BAKR  14,0            * SAVE REGISTER/PSW STATUS
*
* ESTABLISH ADDRESSABILITY FOR THIS PROGRAM.
*
          LR   12,15            * REG 12 BECOMES BASE REGISTER
          DROP  15              *
          USING EXAMPLE,12      * ESTABLISH ADDRESSABILITY
*
* OBTAIN DYNAMIC STORAGE AREA FOR THIS REENTRANT PROGRAM.
*
          L    2,DYNSIZE        * LENGTH TO OBTAIN
          STORAGE  OBTAIN,ADDR=(1),SP=0,LENGTH=(2)
          LR   13,1             * SAVE DYNAMIC AREA ADDRESS
          USING  DYNAREA,13     * ADDRESSABILITY TO DYNAMIC AREA
*
* SET UP THE REMOTE PARAMETER LIST FOR THE ESTAEX MACRO.
*
          MVC  RMTESTAEX(@LSTSIZE),LSTESTAEX
*
* DEFINE AND ACTIVATE AN ESTAEX RECOVERY ROUTINE AT LABEL 'RECOVERY'.
*
          ESTAEX RECOVERY,PARAM=DYNAREA,MF=(E,RMTESTAEX)
*********************************************************************
*
*  CODE FOR THE MAINLINE ROUTINE FUNCTION CAN BE INSERTED HERE
*
*     IF AN ERROR OCCURS IN THE MAINLINE ROUTINE, THEN THE SYSTEM WILL
*     PASS CONTROL TO RECOVERY.
*
*********************************************************************
*
RETRYPT   DS 0H
*********************************************************************
*
*  CODE FOR THE RETRY ROUTINE FUNCTION CAN BE INSERTED HERE
*
*********************************************************************
          ESTAEX 0              * DELETE THE ESTAEX
          LR   1,13             * FREE DYNAMIC AREA, ADDRESS TO FREE
          L    2,DYNSIZE        * LENGTH TO FREE
          STORAGE  RELEASE,ADDR=(1),SP=0,LENGTH=(2)
          PR                    * RESTORE STATUS & RETURN TO CALLER
*********************************************************************
*
* RECOVERY ROUTINE
*
*********************************************************************
RECOVERY  DS    0H              * ENTRY POINT FOR ESTAEX RECOVERY ROUTINE
*
```

```
* HANDLE INPUT FROM THE SYSTEM  AND RE-ESTABLISH ADDRESSABILITY FOR
* BASE REGISTER (12) AND DYNAMIC AREA REGISTER (13)
*
          PUSH  USING
          DROP  ,               * ENSURE NO SPURIOUS USING REFERENCES
          USING RECOVERY,15     * TEMPORARY ADDRESSABILITY
          L     12,#BASE        * RELOAD THE BASE REGISTER
          DROP 15               * RELEASE TEMPORARY ADDRESSABILITY
          USING EXAMPLE,12      * USE THE BASE REGISTER
          USING DYNAREA,13      * DYNAMIC AREA ADDRESSABILITY
          C    0,TESTNOSDWA     * IS THERE AN SDWA PRESENT?
          BE   NOSDWA           * NO, DO NOT USE THE SDWA
HAVESDWA  DS   0H
          USING SDWA,1          * ADDRESSABILITY TO SDWA
          L    13,SDWAPARM      * ADDRESS OF PARAMETER ADDRESS
          L    13,0(13)         * PARAMETER ADDRESS (DYNAREA)
          MVC  SAVE_ABCC,SDWAABCC * SAVE THE COMPLETION CODE
          B    RECOV1           * CONTINUE WITH COMMON RECOVERY
NOSDWA    LR   13,2             * PARAMETER ADDRESS (DYNAREA)
          ST   1,SAVE_ABCC      * SAVE THE COMPLETION CODE
          SR   1,1              * NO SDWA IS AVAILABLE, CLEAR REGISTER
*
* COMMON RECOVERY PROCESSING
*
RECOV1    DS   0H               * COMMON RECOVERY PROCESSING
          ST   1,SAVE_SDWA      * SAVE THE SDWA ADDRESS
          ST   14,SAVE_RETURNR14 * RETURN ADDRESS TO THE SYSTEM
*
*********************************************************************
*
*  CODE FOR THE RECOVERY ROUTINE FUNCTION SHOULD BE INSERTED HERE
*
*********************************************************************
*
* IF THERE IS NO SDWA, THEN SET UP FOR PERCOLATION
*
          L    1,SAVE_SDWA      * RESTORE SDWA REGISTER (1)
          LTR  1,1              * IS THERE AN SDWA?
          BZ   NORETRY          * NO, DO NOT ATTEMPT TO RETRY
*
* CHECK SDWACLUP TO SEE IF RETRY IS ALLOWED
*
          TM   SDWAERRD,SDWACLUP * IS RETRY ALLOWED?
          BNZ  NORETRY          * NO, DO NOT ATTEMPT TO RETRY
*
* SET UP THE RETURN PARAMETERS TO THE SYSTEM.  THE SETRP MACRO UPDATES
* THE SDWA.  NOTE: THE WKAREA PARAMETER DEFAULTS TO REGISTER 1, WHICH
* HAS THE ADDRESS OF THE SDWA.  ALSO NOTE THAT OTHER REGISTERS MIGHT
* NEED TO BE UPDATED TO MEET THE NEEDS OF DIFFERENT PROGRAMS.
*
          ST     12,SDWASR12    * BASE REGISTER 12 FOR RETRY
          ST     13,SDWASR13    * DYNAMIC AREA REGISTER 13 FOR RETRY
          SETRP  RETREGS=YES,RC=4,RETADDR=RETRYPT,FRESDWA=YES
          B    RECOV2           * CONTINUE WITH COMMON RECOVERY
NORETRY   DS  0H                * BRANCH HERE WHEN NOT GOING TO RETRY
          LA  15,0              * RETURN CODE TO INDICATE PERCOLATION
RECOV2    DS  0H                * COMPLETE THE RETURN TO THE SYSTEM
          L   14,SAVE_RETURNR14 * SET THE RETURN ADDRESS TO THE SYSTEM
```

```
         BR  14                * RETURN TO THE SYSTEM
*
* STATIC STORAGE AREA
*
TESTNOSDWA DC  F'12'            * TEST FOR NO SDWA CONDITION
#BASE     DC   A(EXAMPLE)       * BASE REGISTER VALUE
DYNSIZE   DC   AL4(@DYNSIZE)    * DYNAMIC AREA SIZE
LSTESTAEX ESTAEX RECOVERY,MF=L  * LIST FORM OF ESTAEX PARAMETER LIST
@LSTSIZE  EQU *-LSTESTAEX       * SIZE OF ESTAEX PARAMETER LIST
*
* DYNAMIC AREA STORAGE FOR REENTRANT PROGRAM
*
DYNAREA   DSECT                 * DYNAMIC STORAGE
SAVEAREA  DS   18F              * REGISTER SAVE AREA
SAVE_SDWA DS   F                * SDWA ADDRESS ON ENTRY TO RECOVERY
SAVE_ABCC DS   F                * COMPLETION CODE
SAVE_RETURNR14 DS F             * RETURN ADDR. TO SYSTEM FROM RECOVERY
RMTESTAEX DS   CL(@LSTSIZE)     * REMOTE ESTAEX PARAMETER LIST
STATUS    DS   F                * MAINLINE STATUS INDICATOR
@ENDDYN   DS   0X               * USED TO CALCULATE DYNAMIC AREA SIZE
@DYNSIZE  EQU  ((@ENDDYN-DYNAREA+7)/8)*8  * DYNAMIC AREA SIZE
*
* INCLUDE MACROS
*
          IHASDWA
          END
```

Understanding Advanced Recovery Topics

This section contains information about the following advanced recovery topics:

- "Invoking RTM (ABEND Macro)"
- "Providing Multiple Recovery Routines" on page 8-46
- "Providing Recovery for Recovery Routines" on page 8-47
- "Providing Recovery for Multitasking Programs" on page 8-47.

Invoking RTM (ABEND Macro)

Any routine can issue the ABEND macro to direct the recovery termination services to itself (cause entry into its recovery routine) or to its callers. The issuer of ABEND should remove its own recovery routine if it wishes its caller to be ended abnormally or to enter recovery. Control does not return to the issuer of the macro (except as a result of a retry).

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/E, the TSO/E job is 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 the system to end the job step task abnormally.

If you have provided a recovery routine for your program, the system passes control to your routine. If you have not provided a recovery routine, the system handles the problem. The action the system takes depends on whether the job step is going to be terminated.

If the job step is not going to be terminated, the system:

- 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 (TCB) of the active task (the task for which the ABEND macro was issued).
- Posts the event control block (ECB) with the completion code specified in the ABEND macro, if the ECB parameter was coded in the 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 ATTACHX macro issued to create the active task.
- Calls a routine to free the storage of the terminating task's TCB, if neither the ECB nor ETXR parameter were specified by the ATTACHX macro.

If the job step is to be terminated, the system:

- Releases the resources owned by each task, starting with the lowest level task, for all tasks in the job step. The system does not give control to any end-of-task exit.
- 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 define your own recovery routine to perform similar functions and any other functions that your program requires. Use either the ESTAE or ESTAEX macro, or the ATTACHX macro with the ESTAI option to provide a recovery routine that gets control whenever your program issues an ABEND macro. If your program is running in AR ASC mode, use the ESTAEX or ATTACHX macro.

Providing Multiple Recovery Routines

A single program can activate more than one ESTAE-type recovery routine by issuing the ESTAE or ESTAEX macro more than once with the CT parameter. The program can also overlay recovery routines by issuing ESTAE or ESTAEX with the OV parameter, or deactivate recovery routines by issuing ESTAE or ESTAEX with an address of zero.

ESTAE-type recovery routines get control in LIFO order, so the last ESTAE-type recovery routine activated is the first to get control. Remember that ESTAE-type recovery routines include ESTAE and ESTAEX routines, and ESTAI routines. ESTAI routines are entered after all other ESTAE-type recovery routines that exist for a given task have received control and have either failed or percolated.

MVS functions provide their own recovery routines; thus, percolation can pass control to both installation-written and system-provided recovery routines. If all

recovery routines percolate -- that is, no recovery routine can recover from the error -- then the task is terminated.

When a recovery routine gets control and cannot recover from the error (that is, it does not retry), it must free the resources held by the mainline routine and request that the system continue with error processing (percolate). Note that a recovery routine entered with the SDWACLUP bit set to one, indicating that retry is not permitted, has no choice but to percolate. When the recovery routine requests percolation, the previously activated recovery routine gets control. When a retry is not requested and the system has entered all possible recovery routines, the task ends abnormally.

When a recovery routine requests percolation, it is deactivated; that is, it can no longer get control for this error. A deactivated recovery routine is not entered again unless that recovery routine is activated again after a retry.

Providing Recovery for Recovery Routines

In some situations, the function a recovery routine performs is so essential that you should provide a recovery routine to recover from errors in the recovery routine. Two examples of such situations are:

1. The availability of some resources can be so critical to continued system or subsystem operation that it might be necessary to provide a recovery routine for the recovery routine, thus ensuring the availability of the critical resources.
2. A recovery routine might perform a function that is, in effect, an extension of the mainline routine's processing. For example, a system service might elect to check a caller's parameter list for fetch or store protection. The service references the user's data in the user's key and, as a result of protection, suffers a program check. The recovery routine gets control and requests a retry to pass a particular return code to the mainline routine. If this recovery routine ends abnormally and does not provide its own recovery, then the caller's recovery routine gets control, and the caller does not get an opportunity to check the return code that it was expecting.

You can activate an ESTAE-type recovery routine from another ESTAE-type recovery routine. Any recovery routine activated from a recovery routine is called a nested recovery routine. Nested ESTAE or ESTAEX routines can retry; the retry routine runs under the RB of the ESTAE-type recovery routine that activated the nested recovery routine.

Providing Recovery for Multitasking Programs

There are situations where the system does not provide serialization between recovery routines for different TCBs in an address space. When possible you should write your recovery routines so that serialization is not required.

When a recovery routine requires serialization with other TCBs in the address space then the recovery routine must provide its own serialization. Serialization must be carefully designed to avoid causing deadlock situations.

One serialization technique to ensure the order of termination processing is to use the DETACH macro. Issuing DETACH ensures that the detached task and its recovery routines complete before processing for the issuing task proceeds.

DETACH can only be used for tasks that were directly attached by the recovery routine's TCB.

Another important aspect of recovery is releasing resources. Releasing serialization resources (locks, ENQs, latches) in ESTAE-type recovery routines, rather than leaving them to be released by a resource manager, helps avoid deadlocks in recovery processing.

Using STAE/STAI Routines.

Notes:

1. **IBM recommends** you use the ESTAEX or ESTAE macro, or the ESTAI parameter on ATTACHX.
2. Under certain circumstances, STAE or STAI routines might receive control in a restricted environment. See “Restricted Environments” on page 8-38 for more information.

The STAE macro causes a recovery routine address to be made known to the system. This recovery routine is associated with the task and the RB that issued STAE. Use of the STAI option on the ATTACH macro also causes a recovery routine to be made known to the system, but the routine is associated with the subtask created through ATTACH. Furthermore, STAI recovery routines are propagated to all lower-level subtasks of the subtask created with ATTACH that specified the STAI parameter.

If a task is scheduled for abnormal termination, the recovery routine specified by the most recently issued STAE macro gets control and runs under a program request block created by the SYNCH service routine. Only one STAE routine receives control. The STAE routine must specify, by a return code in register 15, whether a retry routine is to be scheduled. If no retry routine is to be scheduled (return code = 0) and this is a subtask with STAI recovery routines, the STAI recovery routine is given control. If there is no STAI recovery routine, abnormal termination continues.

If there is more than one STAI recovery routine existing for a task, the newest one receives control first. If it requests that termination continue (return code = 0), the next STAI routine receives control. This continues until either all STAI routines have received control and requested that the termination continue, a STAI routine requests retry (return code = 4 or 12), or a STAI routine requests that the termination continue but no further STAI routines receive control (return code = 16).

Programs running under a single TCB can issue more than one STAE macro with the CT parameter to define more than one STAE routine. Each issuance temporarily deactivates the previous STAE routine. The previous STAE routine becomes active when the current STAE routine is deactivated.

A STAE routine is deactivated (it cannot receive control again for this error) under any of the following circumstances:

- When the RB that activated it goes away (unless it issued XCTL and specified the XCTL=YES parameter on the STAE macro)
- When the STAE macro is issued with an address of 0
- When the STAE routine receives control.

If a STAE routine receives control and requests retry, the retry routine reissues the STAE macro if it wants continued STAE protection.

A STAI routine is deactivated if the task completes or if the STAI routine requests that termination continue and no further STAI processing be done. In the latter case, all STAI recovery routines for the task are deactivated.

STAE and STAI Routine Environment:

Prior to entering a STAE or STAI recovery routine, the system attempts to obtain and initialize a work area that contains information about the error. The first 4 bytes of the SDWA contains the address of the user parameter area specified on the STAE macro or the STAI parameter on the ATTACH macro.

Upon entry to the STAE or STAI routine, the GPRs contain the following:

If an SDWA was obtained:

GPR	Contents
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 active I/O at abend time. 16 (X'10') — Active I/O, if any, was allowed to continue.
1	Address of the SDWA.
2	Address of the parameter area you specified on the PARAM parameter.
3 - 12	Do not contain any information for use by the routine.
13	Save area address.
14	Return address.
15	Address of STAE recovery routine.

If no SDWA was available:

GPR	Contents
0	12 (X'0C') to indicate that no SDWA was obtained.
1	Completion code.
2	Address of user-supplied parameter list.
3 - 13	Do not contain any information for use by the routine.
14	Return address.
15	Address of STAE recovery routine.

When the STAE or STAI routine has completed, it should return to the system through the contents of GPR 14. GPR 15 should contain one of the following return codes:

Return Code	Action
0	Continue the termination. The next STAI, ESTAI, or ESTAE routine will be given control. No other STAE routines will receive control.
4,8,12	A retry routine is to be scheduled. **Note:** These values are not valid for STAI/ESTAI routines that receive control when a resource manager fails during **normal** termination of a task. See “Restricted Environments” on page 8-38 for more information.

16 No further STAI/ESTAI processing is to occur. This code may be issued only by a STAI/ESTAI routine

For the following situations, STAE/STAI routines are not entered:

- If the abnormal termination is caused by an operator's CANCEL command, job step timer expiration, or the detaching of an incomplete task without the STAE=YES option.
- If the failing task has been in a wait state for more than 30 minutes.
- If the STAE macro was issued by a subtask and the attaching task abnormally terminates.
- If the recovery routine was specified for a subtask, through the STAI parameter of the ATTACH macro, and the attaching task abnormally terminates.
- If a problem other than those above arises while the system is preparing to give control to the STAE routine.
- If another task in the job step terminates without the step option.

STAE and STAI Retry Routines:

If the STAE retry routine is scheduled, the system automatically deactivates the active STAE routine; the preceding STAE routine, if one exists, then becomes activated. Users wanting to maintain STAE protection during retry must reactivate a STAE routine within the retry routine, or must issue multiple STAE requests prior to the time that the retry routine gains control.

Like the STAE/STAI recovery routine, the STAE/STAI retry routine must be in storage when the recovery routine determines that retry is to be attempted. If not already resident in your program, the retry routine may be brought into storage through the LOAD macro by either the mainline routine or the recovery routine.

If the STAE/STAI routine indicates that a retry routine has been provided (return code = 4, 8, or 12), register 0 must contain the address of the retry routine. The STAE routine that requested retry is deactivated and the request block queue is purged up to, but not including, the RB of the program that issued the STAE macro. In addition, open DCBs that can be associated with the purged RBs are closed and queued I/O requests associated with the DCBs being closed are purged.

The RB purge is an attempt to cancel the effects of partially run programs that are at a lower level in the program hierarchy than the program under which the retry occurs. However, certain effects on the system are not canceled by this RB purge. Generally, these effects are TCB-related and are not identifiable at the RB level. Examples of these effects are as follows:

- Subtasks created by a program to be purged. Subtasks cannot be associated with an RB; the structure is defined through TCBs.
- Resources allocated by the ENQ macro. ENQ resources are associated with the TCB and are not identifiable at the RB level.
- DCBs that exist in dynamically acquired virtual storage. Only DCBs in the program, as defined by the RB through the CDE itself, are closed.

If there are quiesced restorable input/output operations (as specified by PURGE=QUIESCE on the macro invocation), the retry routine can restore them in

the same manner as the retry routine from an ESTAE routine. See “Outstanding I/Os at the Time of Failure” on page 8-28.

If an SDWA was obtained upon entry to the STAE/STAI retry routine, the contents of the GPRs are as follows:

GPR	Contents
0	0
1	Address of the first 104 bytes of the SDWA.
2 - 14	Do not contain any information for use by the routine.
15	Address of the STAE/STAI retry routine.

When the storage is no longer needed, the retry routine should use the FREEMAIN macro with the default subpool to free the first 104 bytes of the SDWA.

If the system was not able to obtain storage for the work area, GPR contents are as follows:

GPR	Contents
0	12 (X'0C')
1	Completion code.
2	Address of purged I/O restore list or 0 if I/O is not restorable.
3 - 14	Do not contain any information for use by the routine.
15	Address of the STAE/STAI retry routine.

The retry routine is entered with the same PSW key as the one in the RBOPSW of the retry RB if the RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem program state, and the PKM of that RB does not have authority to keys less than 8.

Otherwise, the PSW key of the retry routine is that of the task in error.

Chapter 9. Dumping Virtual Storage (ABEND, SNAPX, SNAP, and IEATDUMP Macros)

A problem-state and PSW key 8-15 program can request three types of storage dumps:

- An ABEND dump obtained through 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 the SNAPX macro.
- A Transaction dump obtained througfh the IEATDUMP macro.

Figure 9-1 summarizes reasons for selecting an ABEND, SNAP, or Transaction dump.

Figure 9-1. Reasons for Selecting the Type of Dump

Type of Dump	Reasons for Selecting the Type of Dump
ABEND dumps	Use an ABEND dump when abnormally ending a program because of an unpredictable error. The DD statement in the program's job step determines the type of ABEND dump and the dump contents, as follows: **SYSUDUMP ABEND dumps** These are the smallest of the ABEND dumps, containing data and areas only about the failing program. You can either print or view the dump if it's in a SYSOUT data set or on a tape or direct access data set (DASD). **SYSABEND ABEND dumps** These are the largest of the ABEND dumps, containing all the areas in a SYSUDUMP dump and system areas that are used to analyze the processing in the failing program. You can either print or view the dump if it's in a SYSOUT data set or on a tape or DASD. **SYSMDUMP ABEND dumps** These contain data and areas in the failing program, plus some system areas. The SYSMDUMP dumps can be more useful for diagnosis than other ABEND dumps because you can use IPCS to gather diagnostic information. Use IPCS to format, view, and print dumps.
SNAP dumps	Use a SNAP dump to show one or more user-specified areas in the problem-state program while it is running. A series of SNAP dumps can show a storage area at different stages to display a program's processing. For example, you can use SNAP dumps to show fields holding calculations and the counter in a loop to check processing in the loop.
Transaction dumps (IEATDUMP)	Use Transaction dump to show one or more user-specified areas in the problem-state program while it is running. Transaction dump can be more useful for diagnosis than SNAP dumps because you can use IPCS to gather diagnostic information. Use IPCS to format, view, and print dumps.

OS/390 MVS Diagnosis: Tools and Service Aids shows, in detail, the contents of dumps. *OS/390 MVS IPCS User's Guide* describes how to use IPCS. *OS/390 MVS Programming: Authorized Assembler Services Guide* describes macros that authorized programs can use to dump virtual storage.

© Copyright IBM Corp. 1988, 1999

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.

Note: If your program calls a system service (by issuing a macro or callable service), that system service might encounter a user-induced error and end abnormally. Generally, the system does not take dumps for user-induced errors. If you require such a dump, then it is your responsibility to request one in your recovery routine. See Chapter 8, “Providing Recovery” on page 8-1 for information about writing recovery routines.

Obtaining a Symptom Dump

With all ABEND dumps, you will automatically receive a symptom dump through message IEA995I. This symptom dump provides a summary of error information, which will help you to identify duplicate problems. For more information about message IEA995I, see *OS/390 MVS System Messages, Vol 3 (GDE-IEB)*.

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.

Suppressing Dumps That Duplicate Previous Dumps

If your installation is using dump analysis and elimination (DAE), code your program to provide symptom data that DAE can compare with the symptom data from previous dumps. Through this comparison, DAE can reduce the number of duplicate dumps. Another benefit is that the symptom data, which is stored in the DAE data set, provides a consistent set of data for identifying a failure.

DAE suppresses dumps that match a dump you already have. Each time DAE suppresses a duplicate dump, the system does not collect data for the duplicate or write the duplicate to a data set. In this way, DAE can improve dump management by only dumping unique situations and by minimizing the number of dumps.

To perform dump suppression, DAE builds a symptom string, if the data for it is available. If the symptom string contains the minimum problem data, DAE uses the symptom string to recognize a duplicate SVC dump, SYSMDUMP dump, or Transaction dump requested for a software error. When installation parameters request suppression, DAE suppresses the duplicate dump. The following describes DAE processing.

1. **DAE obtains problem data**. DAE receives the data in the system diagnostic work area (SDWA) or from values in a SYMREC parameter on the SDUMP, SDUMPX or IEATDUMP macro that requested the dump.
   - The system supplies system-level data, such as the abend and reason codes, the failing instruction, and the register/PSW difference.
   - The ESTAE routine or the functional recovery routine (FRR) of the failing program supplies module-level information, such as the failing load module name and the failing CSECT name.
2. **DAE forms a symptom string.** DAE adds a descriptive keyword to each field of problem data to form a symptom. DAE forms MVS symptoms, rather than RETAIN symptoms. DAE combines the symptoms for a requested dump into a symptom string.

   The following tables show the required and optional symptoms. SDWA field names are given for the symptoms the failing program must provide to enable dump suppression. The tables have both MVS and RETAIN symptoms so that you can relate the MVS symptoms DAE uses to the RETAIN symptoms you might use to search the RETAIN data base. An MVS symptom string must contain at least five symptoms that are not null. DAE places symptoms into strings in the order shown in the tables.

   Required symptoms are first and must be present.

Symptom	SDWA Field	MVS Keyword	RETAIN Keyword
Name of the failing load module	SDWAMODN	MOD/name	RIDS/name#L
Name of the failing CSECT	SDWACSCT	CSECT/name	RIDS/name

Optional symptoms must follow the required symptoms. DAE needs at least three of these optional symptoms to make a useful symptom string.

Symptom	SDWA Field	MVS Keyword	RETAIN Keyword
Product/component identifier with the component identifier base	SDWACID, SDWACIDB	PIDS/name	PIDS/name
System completion (abend) code		AB/S0hhh	AB/S0hhh
User completion (abend) code		AB/Udddd	AB/Udddd
Recovery routine name	SDWAREXN	REXN/name	RIDS/name#R
Failing instruction area		FI/area	VALU/Harea
PSW/register difference		REGS/hhhhh	REGS/hhhhh
Reason code, accompanying the abend code or from the REASON parameter of the macro that requests the dump		HRC1/nnnn	PRCS/nnnn
Subcomponent or module subfunction	SDWASC	SUB1/name	VALU/Cname

3. **DAE tries to match the symptom string from the dump to a symptom string for a previous dump** of the same type, that is, SVC dumps, with SVC dumps and SYSMDUMP, or Transaction dumps with a previous SYSMDUMP or Transaction dump. When DAE finds a match, DAE considers the dump to be a duplicate.

   Previous symptom strings are kept by DAE in virtual storage. When DAE is started, usually during IPL, DAE selects from the DAE data set symptom strings that were active in the last 180 days; either the string was created for a unique dump within the last 180 days or its dump count was updated within the last 180 days. The selected symptom strings are placed in virtual storage.

   The systems in a sysplex can share the DAE data set to suppress duplicate dumps across the sysplex. While each system in a sysplex can use its own DAE data set, IBM recommends that systems in a sysplex share a DAE data set so that:

   - DAE can write a dump on one system and suppress duplicates on other systems in the sysplex.
   - Only one DAE data set is required, rather than a data set for each system.

4. **DAE updates the symptom strings in storage and, later, in the DAE data set**, if updating is requested.

   - For a unique symptom string, DAE adds a new record. The record contains the symptom string, the dates of the first and last occurrences, the incidence count for the number of occurrences, and the name of the system that provided the string.
   - For a duplicate symptom string, DAE updates the incidence count for the string, the last-occurrence date, and the name of the last system that found the string.

   In a sysplex, changes to the in-storage strings are propagated to the in-storage strings throughout the sysplex.

5. **DAE suppresses a duplicate dump**, if DAE is enabled for dump suppression.

   Note that, if you specify an ACTION of SVCD, TRDUMP, NOSUP, or RECOVERY on a SLIP command, the command overrides DAE suppression

and the system writes the dump. Also, dumps requested by the DUMP operator command are not eligible for suppression.

When DAE does not suppress a dump, the symptom string is in the dump header; you can view it with the IPCS VERBEXIT DAEDATA subcommand. DAE also issues informational messages to indicate why the dump was not suppressed.

DAE suppresses a dump when all of the following are true:

- DAE located in the dump the minimum set of symptoms.
- The symptom string for the dump matches a symptom string for a previous dump of the same type.
- Either of the following is true:
  - The current ADYSETxx parmlib member specifies SUPPRESS for the type of dump being requested and the VRADAE key is present in the SDWA. To set the VRADAE key, a recovery routine issues the following macro:

    ```
    VRADATA KEY=VRADAE
    ```

  - A VRADATA VRAINIT must be done prior to any VRADATA KEY= request in order for the VRA data to be properly processed by both DAE and the SDWA formatter.
  - The current ADYSETxx parmlib member specifies SUPPRESSALL for the type of dump being requested and the VRANODAE key is absent from the SDWA. The VRANODAE key specifies that the dump is not to be suppressed.

The following table shows the effect of the VRADAE and VRANODAE keys on dump suppression when SUPPRESS and SUPPRESSALL keywords are specified in the ADYSETxx parmlib member. For SUPPRESS, the VRANODAE key can be present or absent; the system does not check it. The table assumes that the symptom string from the dump has matched a previous symptom string.

ADYSETxx Option	VRADAE Key in SDWA	VRANODAE Key in SDWA	Dump Suppressed?
SUPPRESS	Yes	N/A	Yes
SUPPRESS	No	N/A	No
SUPPRESSALL	Yes	No	Yes
SUPPRESSALL	No	Yes	No
SUPPRESSALL	No	No	Yes
SUPPRESSALL	Yes	Yes	No

The only way to ensure that a dump is **not** suppressed, regardless of the contents of the ADYSETxx parmlib member, is to specify the VRANODAE key in the SDWA, or DAE=NO on SYMRBLD used to build a symptom record passed to the SDUMPX or IEATDUMP macro with the SYMREC keyword.

References:

- See *OS/390 MVS Diagnosis: Reference* for symptoms and symptom strings.

- See *OS/390 MVS Initialization and Tuning Reference* for the ADYSETxx and IEACMD00 parmlib members.
- See *OS/390 MVS IPCS Commands* for the VERBEXIT DAEDATA subcommand.

Symptoms Provided by a Recovery Routine

DAE attempts to construct a unique symptom string using specific data that your recovery routine can provide in the SDWA or through a symptom record. For an SVC dump, or a Transaction dump, the symptom record is passed in the SDUMPX or IEATDUMP macro. For a SYSMDUMP, place the symptom record in the ABDUMP symptom area.

To provide symptoms for an SVC dump, do one or more of the following in a recovery routine:

- Place data in the SDWA, which is mapped by the IHASDWA mapping macro.
- In an authorized program, provide a symptom record through a SYMREC parameter on the SDUMPX or SDUMP macro. The symptom record is built using SYMRBLD and mapped by the ADSR mapping macro.
- In an authorized or unauthorized program, provide a symptom record through a SYMREC parameter on the IEATDUMP macro. The symptom record is built using SYMRBLD and mapped by the ADSR mapping macro.
- Use a VRADATA macro to place information in the SDWA variable recording area (VRA), which is mapped by the IHAVRA mapping macro. The VRA is used:
  - To supply additional symptoms for the symptom string.
  - To provide *secondary symptoms* that give problem data not given in the SDWA; for example, the identity of the caller of a service.

  Use VRADATA when the standard symptom data is insufficient to determine if a dump for the program is a duplicate.

For information about coding the VRADATA macro to add data to the SDWA, see *OS/390 MVS Programming: Assembler Services Reference*. For more information about symptom records, see Chapter 10, Reporting Symptom Records (SYMRBLD and SYMREC Macros). For more information about recovery routines, see “Writing Recovery Routines” on page 8-11.

VRADATA Macro: Use the VRAREQ key to tell DAE that the symptom is required and the VRAOPT key to tell DAE that the symptom is optional.

A VRADATA macro with VRAREQ adds a symptom following the two required symptoms (see the previous table). For example, to add a symptom, in this case the name of a data set, and to define the symptom as required:

```
VRADATA KEY=VRAREQ,DATA=TAG1,LEN=L'TAG1
VRADATA KEY=VRADSN,DATA=MYDSN,LEN=L'MYDSN
⋮
TAG1   DC  AL2(VRADSN)
MYDSN  DC  C'DEPT27.PAYROLL'
```

A VRADATA macro with VRAOPT adds an optional symptom following the optional symptoms (see the previous table). For example, to add an optional symptom with the data at the address in register 5, and to define the symptom as optional:

```
LA  R5,VRAOPTKEY
VRADATA KEY=VRAOPT,LEN=2,DATA=(5)
VRADAYA KEY=VRACA,DATA=PGMCALLR
⋮
VRAOPTKEY DC AL2(VRACA)
PGMCALLR  DS  A
```

If the symptom is to be the caller's address, the data pointed to would consist of X'003C', which represents the key VRACA.

See *OS/390 MVS Diagnosis: Reference* for the VRADATA keys.

Required Symptom Data: The recovery routine must provide the following minimum data to enable dump suppression by DAE:

SDWA Field	Data	Example
SDWAMODN	Failing module name	IEAVTCXX
SDWACSCT	Failing CSECT name	IEAVTC22
SDWACID	Product or component identifier	SCDMP
SDWACIB	Component identifier base	5655
SDWAREXN	Recovery routine name	IEAVTC2R
SDWASC	Subcomponent or module subfunction	RSM-PGFIX

To obtain the failing module name, the failing CSECT name, and the recovery module name, the recovery routine can set up a RECPARM parameter list and specify it on a SETRP macro. For information, see the RECPARM parameter of the SETRP macro in *OS/390 MVS Programming: Authorized Assembler Services Reference SET-WTO*.

Correct Module and CSECT Names: Obtaining the correct module and CSECT names may be difficult, especially when the PSW does not point within the program areas. Problems can also occur when the program uses the following:

- User exits: When the program calls a user exit, which in turn invokes system services, such as getting a lock, the recovery routine often cannot identify the exit as the failing module and CSECT. To avoid this problem, the program should save the name of the user exit, so that the recovery routine can use the saved name.
- Common recovery routines: The program should maintain footprints or require that all modules using a common recovery routine update a common parameter list with the name of the current module, CSECT, subfunction, and so on. The recovery routine can obtain data from the common parameter list when filling in the SDWA.

Symptoms: When you provide symptom information, select each symptom carefully. If a symptom is too precise, no other failure will have that symptom; if the symptom is too general, many failures will have the same symptom. For example, do not use addresses as symptoms; instead, use names of modules and components.

DAE accumulates up to 20 specified required and optional symptoms and up to 20 additional symptoms, if specified. The maximum string length is 150, so that not all

of the additional symptoms may appear in the string. A recovery routine can change the minimum number of symptoms and the minimum string lengths that DAE is to use for symptom matching for a particular dump. To make these changes, code the following VRADATA macro keys in the recovery routine:

- The VRAMINSC key controls the minimum number of symptoms.
- The VRAMINSL key controls the minimum string length.

Control of Suppression: When the ADYSETxx parmlib member being used by the installation contains SUPPRESS, a recovery routine must indicate that enough data is available to suppress a duplicate dump. To indicate to DAE that the SDWA contains enough data, set the VRADAE indicator in the SDWA variable recording area by issuing the following:

```
VRADATA KEY=VRADAE
```

If the recovery routine cannot provide enough data in the SDWA suppression, the recovery routine should indicate that its dump is not eligible for suppression, even when the ADYSETxx member contains SUPPRESSALL. The routine should set the VRANODAE indicator by issuing the following:

```
VRADATA KEY=VRANODAE
```

The VRANODAE key is useful for error environments that generate identical symptoms but represent different problems.

When a Dump is Not Suppressed

When DAE is active but does not suppress a dump, DAE adds the reason that the dump is not suppressed to the dump header record. When viewing a dump online or printing a dump, specify the IPCS VERBEXIT DAEDATA subcommand to see the reason that a dump was not suppressed.

Some reasons for not suppressing a dump are:

- The dump is unique. DAE found no match for the symptom string.
- The current ADYSETxx member does not specify SUPPRESS or SUPPRESSALL for the type of dump being requested.
- A SLIP operator command specifies that the dump is not to be suppressed. A SLIP command with ACTION=SVCD, ACTION=SYNCSVCD, ACTION=STDUMP, ACTION=RECOVERY, or ACTION=TRDUMP always produces a dump. ACTION=NOSUP stops suppression.
- DAE could not find all required symptoms.
- DAE could not find the minimum number of symptoms.
- The symptom string built by DAE was shorter than the minimum length.
- DAE found certain errors. For example, a required symptom had a key that was not valid.
- The VRADAE key in the SDWA is absent, signifying that the dump should not be suppressed, and the DAE parameter of the ADYSETxx parmlib member does not specify SUPPRESSALL.
- The VRANODAE key is present in the SDWA, specifying that the dump should not be suppressed.

SNAP Dumps

A program can request a SNAP dump at any time during its processing by issuing a SNAPX or 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. Ensure 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 (DCB) 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 DCB must contain the following parameters: DSORG=PS, RECFM=VBA, MACRF=W, BLKSIZE=882 or 1632, and LRECL=125. (The DCB is described in *DFSMS/MVS Using Data Sets*, and *DFSMS/MVS Macro Instructions for 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 DCB.

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 for an ABEND or SNAP 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 *OS/390 MVS Recovery and Reconfiguration Guide*.

Transaction Dumps

Transaction dump is a service used to request an unformatted dump of virtual storage to a data set, similar to a SYSMDUMP. It is invoked with the IEATDUMP assembler macro which issues SVC 51. You can request that the dump be written to a data set that is either pre- allocated or automatically allocated. To request a pre-allocated data set, specify an open MACRF=W DCB that contains sufficient space in one or more extents for the entire dump to be written. If you don't provide a large enough data set, you will receive a partial dump. To request automatic allocation, ensure a dump will not be truncated due to space constraints and specify a data set name pattern. Automatic allocation is done to SYSALLDA.

When a Transaction dump is written, a dump directory record describing the dump may be written. Specify the dump directory to be used with the IDX keyword on the dump request. If you do not specify a dump directory, the directory allocated to IPCSDDIR in the current job step will be used. If no dump directory is specified and IPCSDDIR is not allocated, no record describing the dump will be written.

Dump suppression occurs using symptoms available in the current SDWA or a symptom string may be provided (via the SYMREC keyword). If you provide a symptom string, and an SDWA exists, the symptom string is used for suppression purposes. Statistics for dump suppression are contained in the DAE data set and are not differentiated from SYSMDUMPs.

Authorized users may specify the REMOTE keyword on the IEATDUMP macro to request other address spaces on the current or other MVS images (in the same sysplex) be dumped. When you request remote dumps, automatic allocation must also be used.

Transaction dump uses an incident token to associate this dump with other diagnostic information. Automatic allocation also uses this incident token for symbol substitution in the data set name pattern. You can generate an incident token using the IEAINTKN macro and providing the INTOKEN keyword on the dump request. If you do not provide an incident token, one will be generated and used internally. While an incident token may always be specified, it may be especially important when remote dumps are requested.

An asynchronous dump may be requested by specifying ASYNC=YES on the dump request. It is recommended that an ECB be specified for asynchronous dumps to ensure that any volatile resources are captured before being freed.

Chapter 10. Reporting Symptom Records (SYMRBLD and SYMREC Macros)

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 to the logrec data set, the online repository where MVS collects error information. Programmers can analyze the records in the logrec data set to diagnose and debug problems in the installation's applications.

The unit of information stored in the logrec data set 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 structured data base (SDB) format.

An installation's programmers can do the following:

- Build the symptom records using the SYMRBLD macro.
- Record the symptom records on the logrec data set using SYMRBLD or SYMREC.
- Format the symptom records into various reports using EREP and IPCS.

Writing Symptom Records to Logrec Data Set

Your application can build and write symptom records to the logrec data set one of two ways:

- Through invoking the SYMRBLD macro services
- By filling in fields of the ADSR mapping macro, then invoking SYMREC.

Using the SYMRBLD Macro: 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 use ADSR and SYMREC directly.

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 the logrec data set.

For more information about SYMRBLD and SYMREC, see *OS/390 MVS Programming: Assembler Services Reference*.

Using EREP or IPCS: The Environmental Record Editing and Printing (EREP) program processes and presents software error records in the following reports:

- Detail edit report for an abend
- Detail edit report for a symptom record
- System summary report
- Event history report
- Detail summary report

© Copyright IBM Corp. 1988, 1999

EREP User's Guide describes how to use EREP.

IPCS formats the software error records. You can use the IPCS VERBEXIT LOGDATA subcommand to format and print or view the logrec data set records in a dump. For more information about the subcommand, see *OS/390 MVS IPCS Commands*.

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 *OS/390 MVS Data Areas, Vol 1 (ABEP-DALT)*, 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. Examples of typical prefixes are:

Prefix	Data
PIDS/	A component name
RIDS/	A routine name
AB/	An abend code
PRCS/	A return code

For more information about the prefixes and all valid SDB key names, see *OS/390 MVS Programming: Assembler Services Reference*. For a full explanation of symptom strings and how to format and print the four basic symptom record reports, see *OS/390 MVS Diagnosis: Procedures* and *OS/390 MVS Diagnosis: Tools and Service Aids*.

Building a Symptom Record Using the SYMRBLD Macro

Input to the SYMRBLD macro is a storage area for the symptom record, and the diagnostic data for sections 2.1, 3, 4, and 5 of the symptom record. The SYMRBLD macro must be invoked several times to build a complete symptom record. The following describes the sequence:

1. Invoke SYMRBLD with the INITIAL parameter to initialize sections 1 and 2, and provide application data for section 2.1.
2. Invoke SYMRBLD with the PRIMARY parameter to store symptoms into section 3. You may invoke this parameter more than once for one error.
3. Optionally invoke SYMRBLD with the SECONDARY parameter to store symptoms into section 4.
4. Optionally invoke SYMRBLD with the VARIABLE parameter to store data into section 5.
5. Invoke SYMRBLD with the COMPLETE parameter to set the lengths of sections 3, 4, and 5 in section 2.1 and optionally code SYMRBLD to invoke the SYMREC macro for recording to the logrec data set. If you do not code SYMRBLD to invoke the SYMREC macro, your records will not be recorded to the logrec data set.
6. Invoke SYMRBLD with the RESET parameter to rebuild the symptom record using the same storage and application information that was specified using the INITIAL parameter. The RESET parameter is useful when the primary, secondary, and variable sections of the symptom record are to be changed but the application information in section 2.1 remains the same.

Building a Symptom Record Using the ADSR and SYMREC Macros

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.
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 the logrec data set, 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. The architecture level is always 10.

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, which is always 10. 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-F0FFN	Not assignable
F100-FEFF	Reserved
FF00	Any program and non-printable hex data
FF01-FFFF	Not assignable

Chapter 11. 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.

The 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 11-6 and Chapter 12, 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 15, Data Spaces and Hiperspaces.

Note: If your job step is to be executed as a non-pageable (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 11-5.

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

© Copyright IBM Corp. 1988, 1999

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 the GETMAIN macro. Each method, which you select by coding a parameter, has certain advantages.

You can specify the 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.)

To request virtual storage that can be above 16 megabytes, use LOC=ANY. To request virtual storage below 16 megabytes, use LOC=BELOW.

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, and if the subpool is supported below 16 megabytes, GETMAIN allocates the area from virtual storage below 16 megabytes.

The element, variable, and list type of methods do not produce reenterable coding unless coded in the list and execute forms. (See “Implicit Requests for Virtual Storage” on page 11-11 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.

To request virtual storage at a specific address, use LOC with the EXPLICIT parameter and specify the address desired on the INADDR parameter. When you specify EXPLICIT on a request for storage from the same virtual page as previously requested storage, you must request it in the same key, subpool, and central storage areas as on the previous storage request. For example, if you request virtual storage backed with central storage below 16 megabytes, any subsequent requests for storage in that virtual page must be specified as LOC=(EXPLICIT,BELOW).

The virtual storage address specified on INADDR and the central storage backing specified on LOC=EXPLICIT must be a valid combination. For example, if the address specified on INADDR is for storage above 16 megabytes, you must specify LOC=EXPLICIT or LOC=(EXPLICIT,ANY). The following combinations are valid:

- virtual any, central any
- virtual below, central below
- virtual below, central any

For more information, see the description of the GETMAIN macro in *OS/390 MVS Programming: Assembler Services 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 11-1 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 11-1. Example of Using the GETMAIN Macro

The code shown in Figure 11-1 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.

Note: If you issue GETMAIN for less than a single page and you then issue the PGSER macro with the PROTECT option, the entire page is made read-only, whether you have issued GETMAIN for it or not. **IBM recommends** that you use PROTECT for full pages only. This usage avoids making other areas of storage read-only unintentionally. If you update the page using real addresses after the page has been protected, the page must be fixed until it is unprotected; otherwise data might be lost.

Obtaining Storage Through the STORAGE Macro

There are several ways of 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 completes, 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 come 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 request virtual storage at a specific address by using EXPLICIT on the LOC parameter and specifying the address on the INADDR parameter. 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 11-2.

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 *OS/390 MVS Programming: Assembler Services 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 11-6. Implicit releasing of virtual storage is described under “Freeing of Virtual Storage” on page 11-13.

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.**

Note: FREEMAIN can free a page that has been protected through the PGSER macro with the PROTECT option.

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 12, Callable Cell Pool Services describes these services. “Comparison of CPOOL Macro and Callable Cell Pool Services” on page 12-1 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 11-2 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.

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.

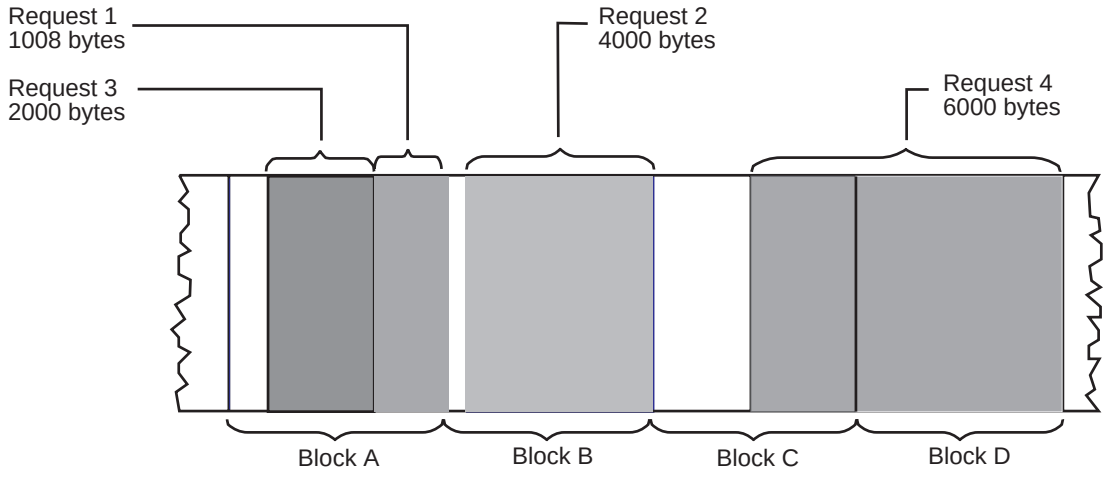


Figure 11-2. Virtual Storage Control

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.

FREEMAIN or STORAGE RELEASE macros can be issued to release any complete subpool except subpool 0, thus releasing complete 4096-byte blocks.

Subpool Characteristics: Problem-state programs running under PSW key 8-15 can specify subpool numbers 0-127, 131, and 132. Subpools 0-127 are task related, meaning that when a task terminates, the system automatically releases any of the subpools from 0 through 127 that are unshared and are associated with the task. Subpools 131 and 132 are job-step related; the system does not release these subpools until the job-step task terminates, even if the task that created these subpools has terminated. All the subpools are pageable, and all are fetch protected except subpool 132.

Storage Keys for Subpools: The storage key for storage in subpools 0-127 is from the TCB associated with the first GETMAIN, STORAGE OBTAIN, or CPOOL BUILD request. All subsequent requests use this key, regardless of the key currently in the TCB.

For subpools 131 and 132, the system assigns storage keys differently, depending on which macros and parameters you use to allocate or free storage. The following table shows how the storage keys are assigned for subpools 131 and 132:

Macros and Parameters	Storage Key
• GETMAIN with LC, LU, VC, VU, EC, EU, or R • FREEMAIN with LC, LU, L, VC, VU, V, EC, EU, E, or R • STORAGE OBTAIN or RELEASE; CALLRKY=YES is specified	The storage key equals the caller's PSW key. (The KEY parameter is ignored.)
• GETMAIN with RC, RU, VRC, VRU • FREEMAIN with RC, RU • CPOOL BUILD	The storage key is the key specified by caller on the KEY parameter. If KEY is not specified, the default equals the caller's PSW key.
• STORAGE OBTAIN or RELEASE; CALLRKY=YES is omitted or CALLRKY=NO is specified	The storage key is the key specified by the caller on the KEY parameter. If KEY is not specified, the default is storage key 0.

A program can issue a request to obtain or release storage from subpool 131 or 132 in a storage key that does not match the PSW key under which the program is running. However, the system will accept the storage request only if the requesting program is authorized to access the storage. To access storage in subpool 131 or 132, a problem-state program that is not APF-authorized and is running under PSW key 8-15 must be able to switch its PSW key to match the storage key. Such a program can switch its PSW key only if a supervisor-state program has previously set up the PSW-key mask (PKM) to enable the PSW key switch. For STORAGE RELEASE or FREEMAIN requests to release all the storage in subpool 131 or 132, the requesting program must be able to switch its PSW key to match all the storage keys that exist in the subpool. For information about the function and structure of the PSW key-mask, and information about switching the PSW key, see *Principles of Operation*.

Owning and Sharing Subpools: 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 and STORAGE macros, you can specify the SP parameter to request storage from subpools 0-127, 131, or 132. If you omit the SP 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 non-reenterable and non-reusable.

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 *OS/390 MVS Programming: Assembler Services 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 11-3 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 11-3. Using the List and the Execute Forms of the DEQ Macro

Non-reenterable 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 non-reenterable and non-reusable.
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 12. 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 12-1 on page 12-3 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/CSRPGT1 and CSRPRGT/CSRPRGT1 services)
- Return a cell to the cell pool (CSRPFRE/CSRPFR1 and CSRPRFR/CSRPRFR1 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.

© Copyright IBM Corp. 1988, 1999

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.

- 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.

Figure 12-1 on page 12-3 illustrates the anchor and extents in Data/Address Space A and the cell storage in Data/Address Space B.

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.

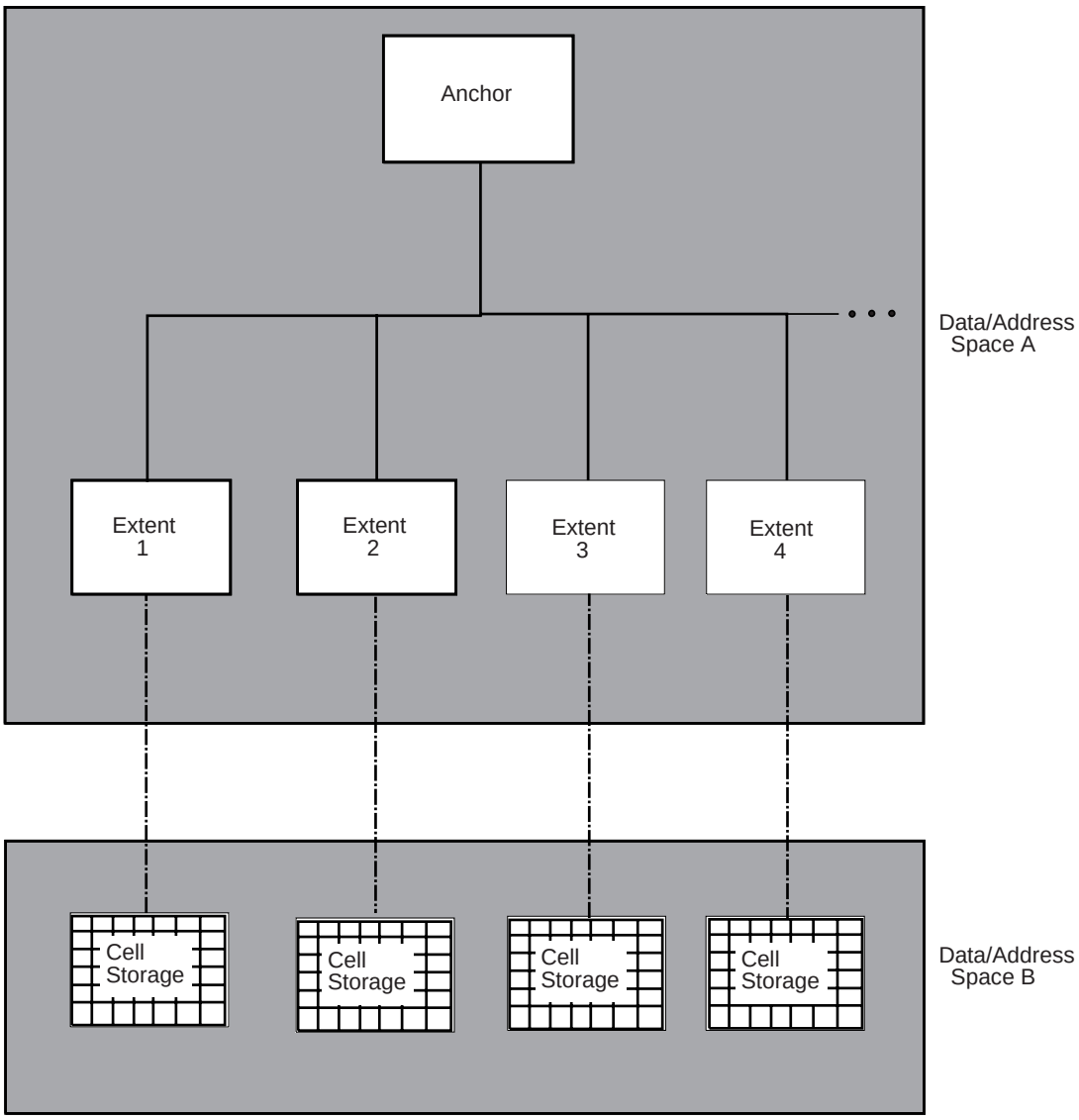


Figure 12-1. Cell Pool Storage

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 15-17 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=HEWL,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 CSRPGT1 (standard CALL interface) and CSRPRGT1 (register interface) services provide the same function with slightly improved performance.

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. The CSRPFR1 (standard CALL interface) and CSRPRFR1 (register interface) services provide the same function with slightly improved performance.

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 13. 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 15-22 and “Using Data-in-Virtual with Hiperspaces” on page 15-37 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.

An application written for data-in-virtual might also benefit by using the IARVSERV macro to share virtual storage, when that storage is in an address space or data space. For information about sharing data in virtual storage through IARVSERV, particularly the restrictions for using the data-in-virtual MAP and UNMAP services, see Chapter 19, “Sharing Data in Virtual Storage (IARVSERV Macro)” on page 19-1.

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.

© Copyright IBM Corp. 1988, 1999

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 15, “Data Spaces and Hiperspaces” on page 15-1 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).

When creating a linear data set for DIV, you can use the LOCVIEW parameter of the DIV macro in conjunction with the other SHAREOPTIONS. LOCVIEW is described on page 13-9. For a complete explanation of SHAREOPTIONS, see *DFSMS/MVS Using 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 *DFSMS/MVS Access Method Services for ICF* or *DFSMS/MVS Summary of Access Method Services for ICF*.

Using the Services of Data-in-Virtual

Each invocation of the DIV macro requests any one of the services provided by data-in-virtual:

- IDENTIFY
- ACCESS
- MAP
- SAVE
- SAVELIST
- 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 13-9.

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.
4. You can map virtual storage pages that are protected only when you specify RETAIN=YES. When the system establishes the virtual window, you can use the PGSER PROTECT macro to protect the data in the window. However, you

must ensure that the data in the window is *not* protected when you issue the RESET form of the DIV macro.

Save, Savelist, and Reset

After using 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 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.

Optionally, SAVE accepts a user list as input. To provide a user list, the application uses the SAVELIST service. SAVELIST returns the addresses of the first and last changed pages in each range of changed pages within the window. The application can then use these addresses as the user list for SAVE. The SAVE operation can be more efficient when using the list of addresses, so an application can improve its performance by using SAVELIST and then SAVE.

When specifying a user list and when a data space or hiperspace contains the window, the caller must use an STOKEN with SAVE to identify the data space or hiperspace.

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.

Note: If you do not want to retain the data in the virtual window, use the PGSER UNPROTECT macro to “unprotect” any protected pages in the window, before you use the UNMAP service.

If you issue UNMAP with RETAIN=NO and there are protected pages in the virtual storage window, the system loses the data in the protected pages and preserves the protection status. If you try to reference the protected pages, the system issues abend X'028'. To access the protected pages again, remove the protection status. Then issue the PGSER RELEASE or DSPSERV RELEASE macro to release all physical paging resources.

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 two required parameters: ID and MODE, and two optional parameters: 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. 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 13-1 on page 13-11.

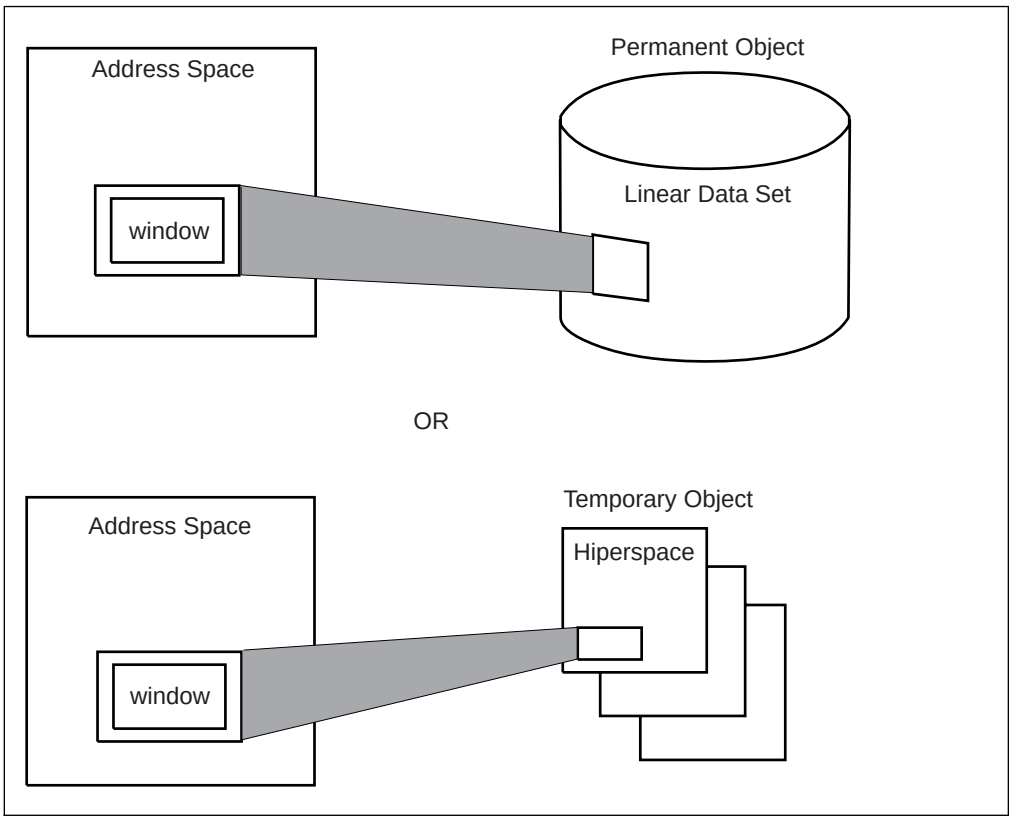


Figure 13-1. Mapping from an Address Space

If your window is in a data space or a hiperspace, you can map only a linear data set. See Figure 13-2.

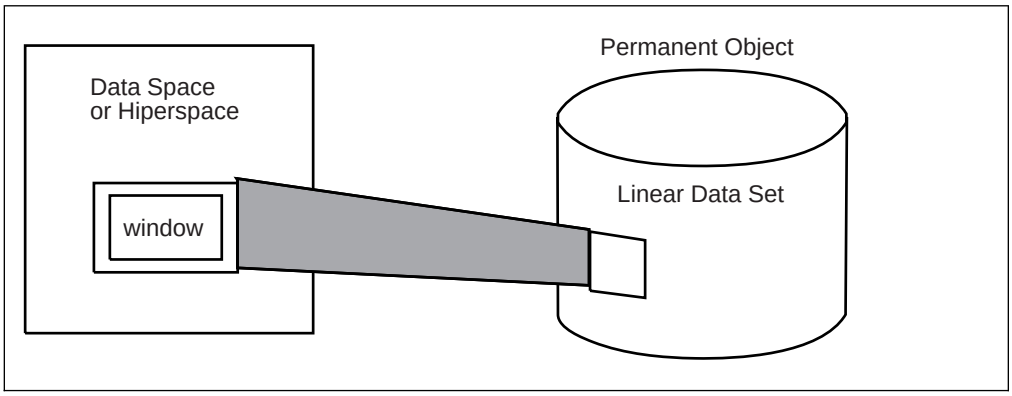


Figure 13-2. Mapping from a Data Space or Hiperspace

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 13-3 on page 13-13.

The MAP service has two required parameters: ID and OFFSET, and five optional parameters: 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.

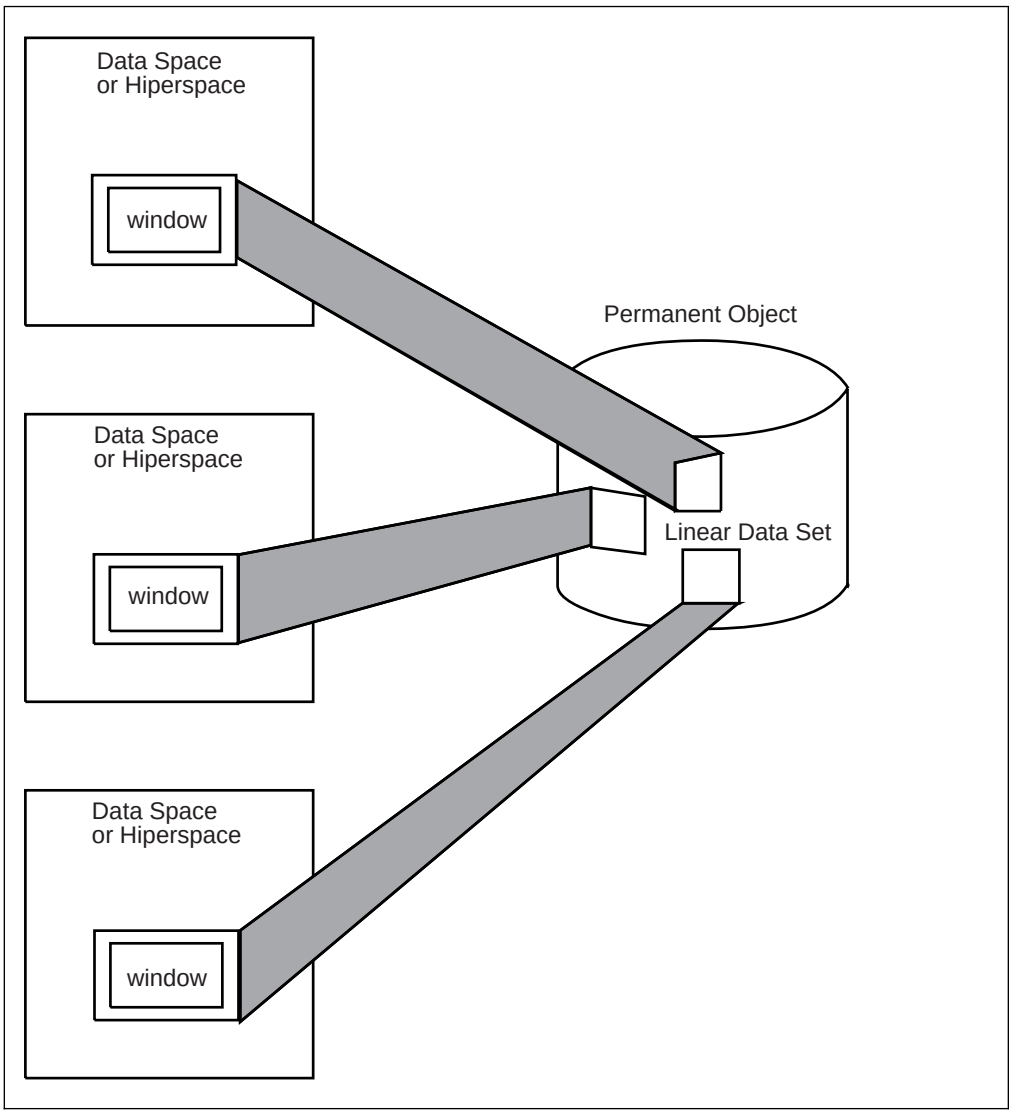


Figure 13-3. Multiple Mapping

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.

Note: Data-in-virtual always allocates an extra block beyond the requested block number range to ensure that there is enough space for an end-of-file (EOF)

record. Therefore, when a DIV object is created without extents, the largest possible span value is the total number of blocks contained in the DIV object minus one.

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 cause your program to abend. Subsequent attempts to reference the mapped virtual space will 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 18-5.

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 one of the following:

- A single and continuous range of blocks in the data-in-virtual object with the use of OFFSET and SPAN. Any virtual storage windows inside this range are eligible to participate in the save.
- A user list supplied through the use of LISTADDR and LISTSIZE. The list must contain the addresses of the first and last changed pages for each range of changed pages within the window. The SAVELIST service can provide these addresses for the user list.

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. You cannot specify OFFSET and SPAN when you specify LISTADDR and LISTSIZE.

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 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.

LISTADDR: The LISTADDR parameter specifies the address of the first entry in the user list. Use this parameter and the LISTSIZE parameter when you specify a user list as input for SAVE.

LISTSIZE: The LISTSIZE parameter specifies the number of entries in the user list. Use this parameter and the LISTADDR parameter when you specify a user list as input for SAVE.

STOKEN: If you specify a user list as input for SAVE and a data space or hiperspace contains the window, you must specify STOKEN when you invoke SAVE. 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.

The SAVELIST Service

The advantage of using SAVELIST with SAVE is improved performance, especially for applications that manipulate graphic images. The SAVELIST service allows the application to inspect and verify data only in pages that have been changed. In a user list provided by the application, SAVELIST returns the addresses of the first and last page in each range of changed pages within the window. The mapped ranges may be either address spaces, data spaces or hiperspaces. If more than one data space or hiperspace is mapped onto a DIV object, the selected range must be contained within a single data space or hiperspace.

The application must set up a user list before issuing SAVELIST. Upon return from SAVELIST, the first word of each list entry holds the virtual storage address of the first page in a range of changed pages. The second word of the entry holds the virtual storage address of the last changed page in that range. In the last valid entry of the user list, the high-order bit of the first word is set to one.

If the reason code indicates that there are more changed pages that can fit in this list, the first word of the last entry in the list contains an offset (in block number format) into the DIV object from which more changed pages might exist. The second word of the last entry contains the span from the new offset to the block pointed to by the original OFFSET/SPAN combination. If more changed pages can fit in the user list, you can issue SAVE with the current list, and then issue SAVELIST and SAVE again to obtain the additional changed pages and to save them.

ID: Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY, which connects a particular user with a particular object.

LISTADDR: The LISTADDR parameter specifies the address of the first entry in the user list.

LISTSIZE: The LISTSIZE parameter specifies the number of entries in the list. The size of the list must be a minimum of three entries and a maximum of 255 entries. The SAVELIST service can place addresses in all but the last two entries, which the macro uses as a work area.

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 two required parameters: ID and AREA, and two optional parameters: 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 15-22
- "Using Data-in-Virtual with Hiperspaces" on page 15-37

General Program Description

This is a description of the program shown in "Data-in-Virtual Sample Program Code" on page 13-23.

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.
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.
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

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:
   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.
   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.
   c. The program acquires storage using GETMAIN to hold the increased size of the object, and issues a DIV MAP for this storage.
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

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.

E function — End the program

7. 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.
8. 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.
9. 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 13-3.

```
//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 14. Using Access Registers

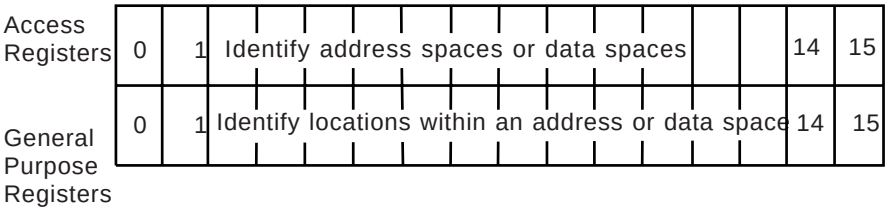
For storing data, MVS offers a program the use of a virtual storage area called a data space. Assembler instructions (such as Load, Store, Add, and Move Character) manipulate the data in a data space. When you use instructions to manipulate data in a data space, your program must use the set of general 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.

© Copyright IBM Corp. 1988, 1999

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 14-1 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.

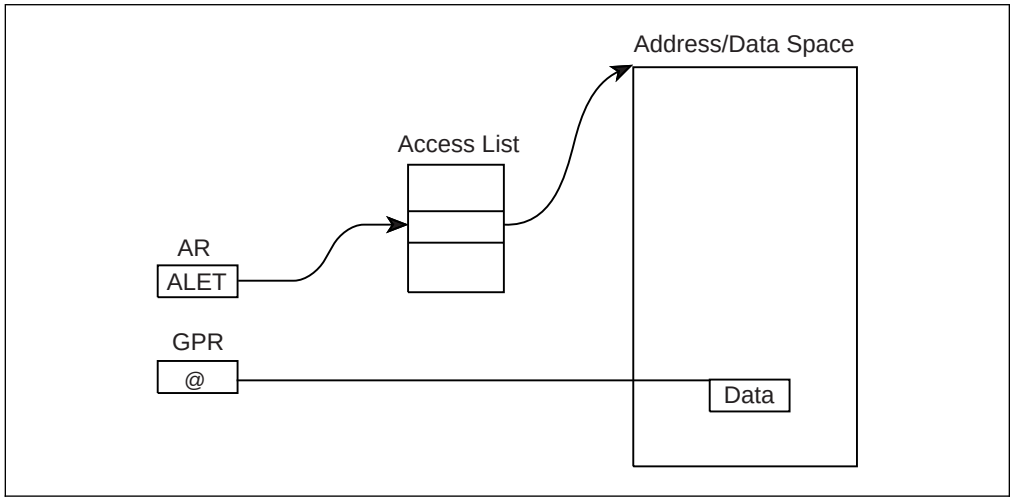


Figure 14-1. Using an ALET to Identify an Address Space or a Data Space

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 14-9.

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 15-31.
  - 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 15-28.

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 *OS/390 MVS Programming: 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 14-2 on page 14-4 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.

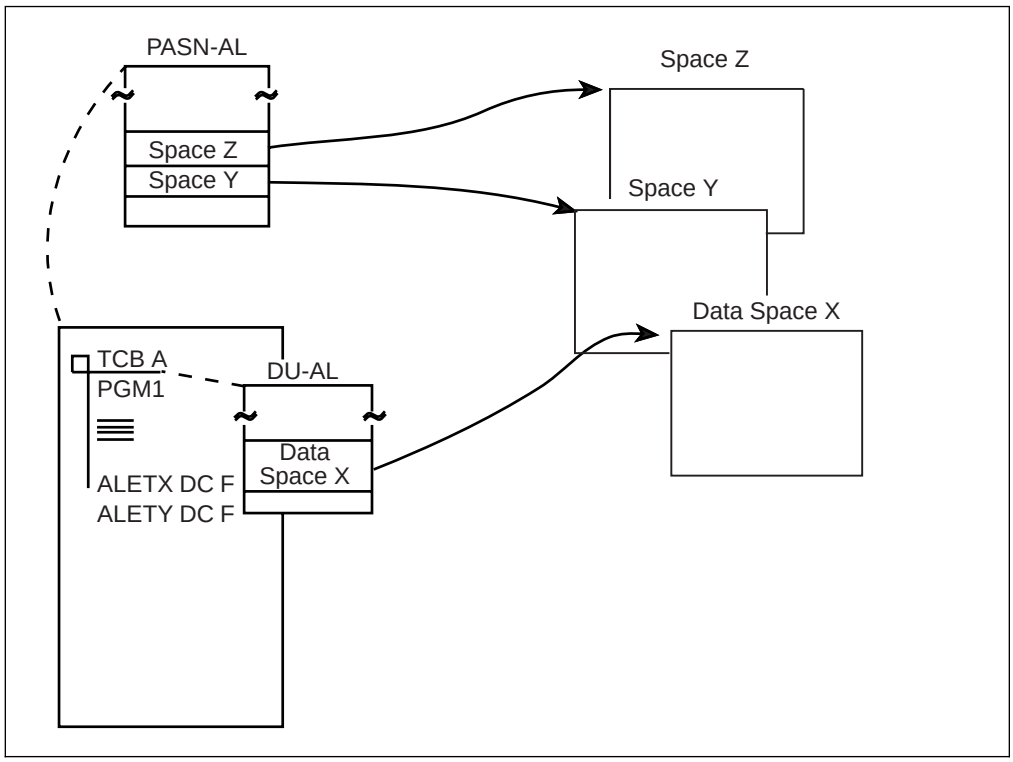


Figure 14-2. An Illustration of a DU-AL

The differences between a DU-AL and a PASN-AL are significant and you need to understand them. The following table summarizes the characteristics of DU-ALs and PASN-ALs as they relate to **problem state programs with PSW key 8 - F**.

Figure 14-3. Characteristics of DU-ALs and PASN-ALs

DU-AL	PASN-AL
Each work unit (TCB and SRB) has its own unique DU-AL. All programs associated with that work unit can use its DU-AL.	Each address space has its own unique PASN-AL. All programs that run in the primary address space can use its PASN-AL.
A program that the work unit represents can add and delete entries on the work unit's DU-AL for the data spaces it created or owns.	A program can add entries for the data spaces it owns or created to the PASN-AL, providing an entry for the data space is not already on the PASN-AL through the actions of another problem state program with PSW 8 - F. A program can delete entries for data spaces it owns or created.
A program cannot pass its task's DU-AL to a program running under another task, with one exception: when a program issues an ATTACH macro, it can pass a copy of its DU-AL to the subtask. This allows the subtask to start with a copy of the attaching task's DU-AL. After the attach, the attaching task and the subtask can add and delete entries on their own DU-ALs.	A PASN-AL cannot be passed from one address space to another.
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.
When the work unit terminates, the DU-AL is purged.	When the owning jobstep task terminates, the PASN-AL is purged.

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 14-4 on page 14-6. 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:

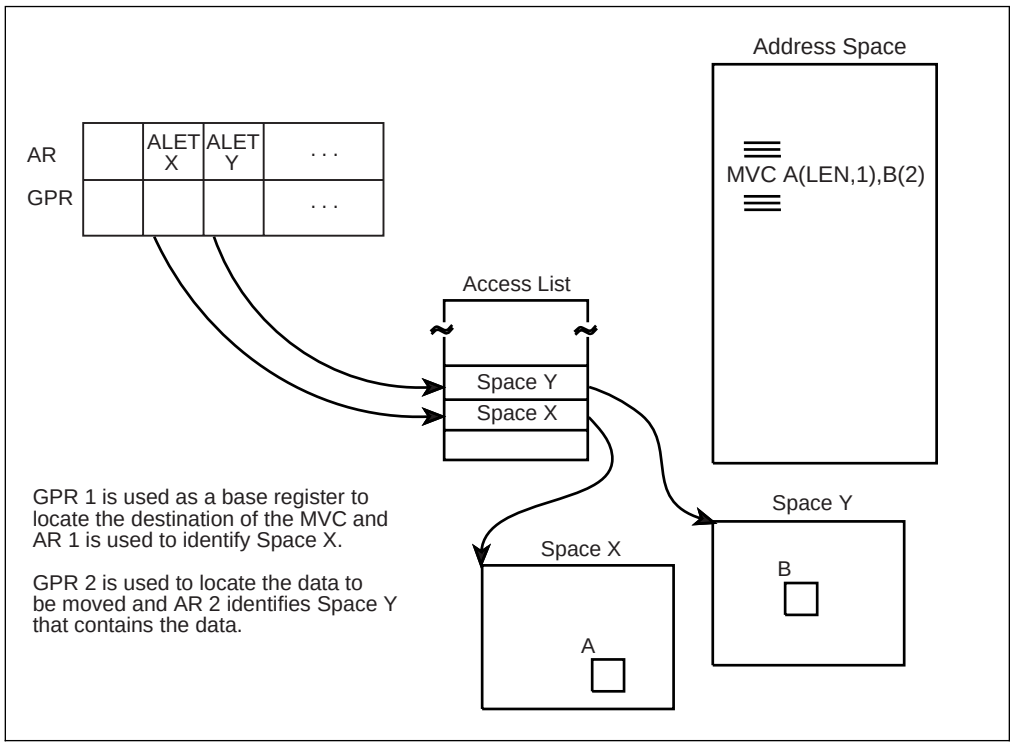


Figure 14-4. Using Instructions in AR Mode

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 14-8 shows several examples of loading a value of zero in an AR.

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 14-5 shows four examples of how a misplaced comma can change how the processor resolves addresses on the load instruction.

Figure 14-5. 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 14-5:

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 14-5:

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 15-31. 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 *OS/390 MVS Programming: Assembler Services 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 *OS/390 MVS Programming: Assembler Services 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.

IBM recommends you always use ESTAEX unless your program and your recovery routine are in 24-bit addressing mode, in which case, you should use ESTAE.

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 *OS/390 MVS IPCS Commands* for more information about the ARCHECK subcommand.

Chapter 15. 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?

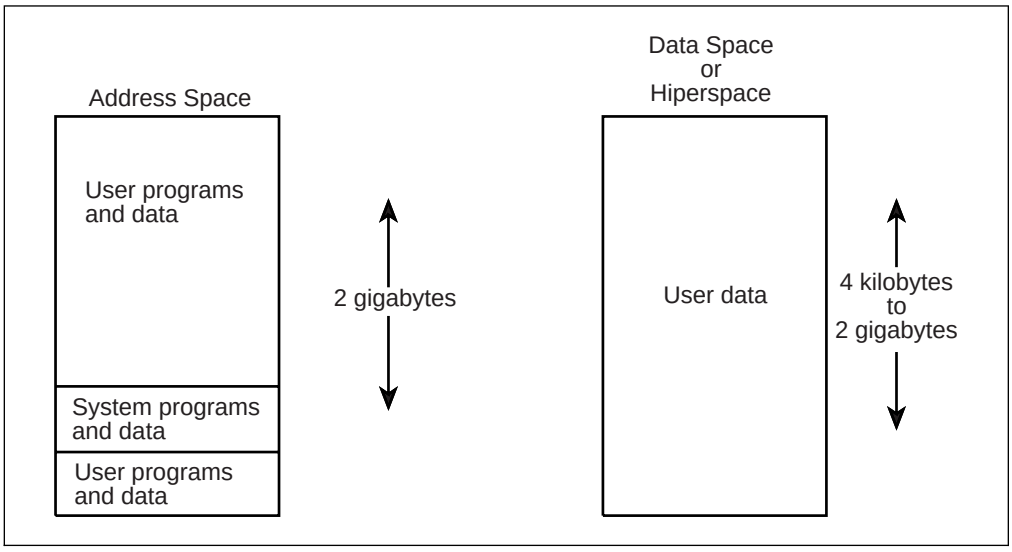
The first part of the chapter helps you make these decisions. Then, if you decide that one of these virtual storage areas would benefit your program, turn to one of the following sections for the information you need to create, use, and delete the area:

- "Creating and Using Data Spaces" on page 15-7
- "Creating and Using Hiperspaces" on page 15-25

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.

© Copyright IBM Corp. 1988, 1999

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 (TYPE=BASIC) or a hiperspace (TYPE=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.

OS/390 MVS Programming: Assembler Services 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 15-17 describes the use of callable cell pool services for data spaces. Information on how to code the services is in Chapter 12, 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 15-1 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.

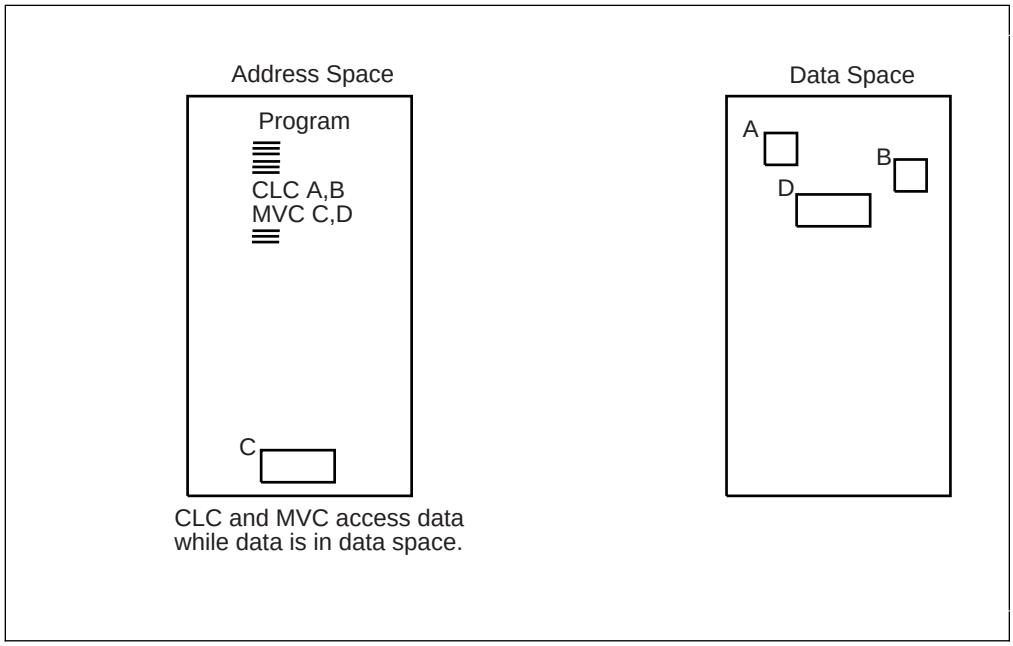


Figure 15-1. Accessing Data in a Data Space

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 15-2 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.

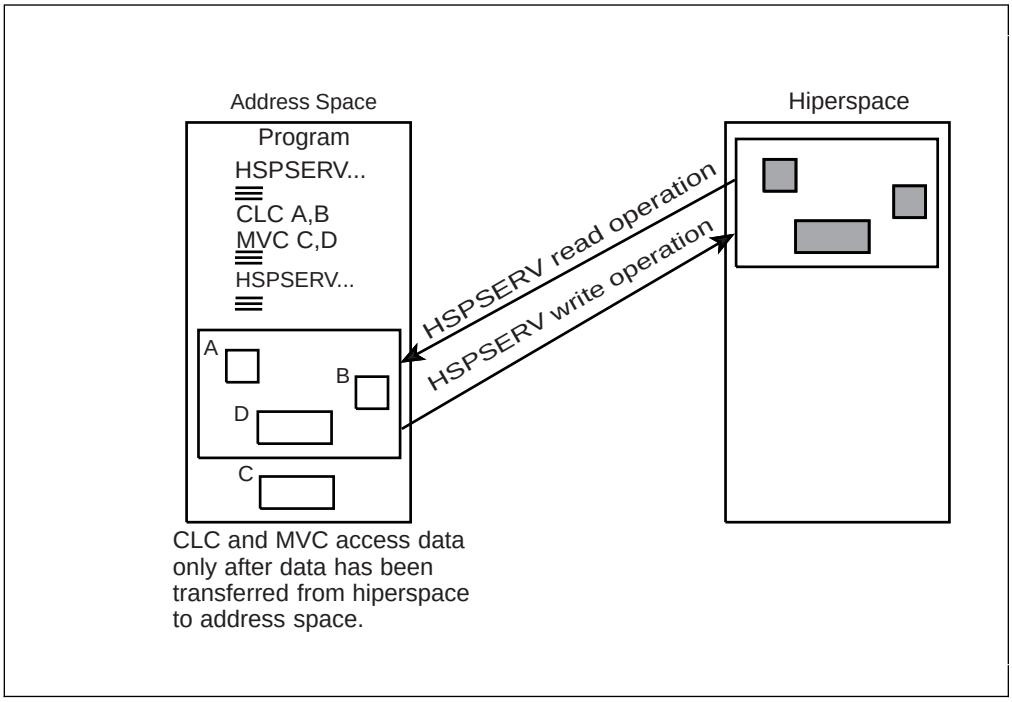


Figure 15-2. Accessing Data in a Hiperspace

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 14, “Using Access Registers” on page 14-1 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 15-3. 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 page areas into central storage from a data space created by any other task in that address space.
OUT	Can page areas out of central storage to a data space created by any other task in that address space.

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 *OS/390 MVS Programming: 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 15-9.

- 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 15-10.
- 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 15-11.
- 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 15-19 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 *OS/390 MVS Programming: Assembler Services 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 PAY␢␢␢␢ 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 J␢␢␢␢␢␢␢ is the name you supply on the NAME parameter and you code GENNAME=YES, the system generates the following name:

```
ncccJ
```

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 15-3.

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 15-4 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
```

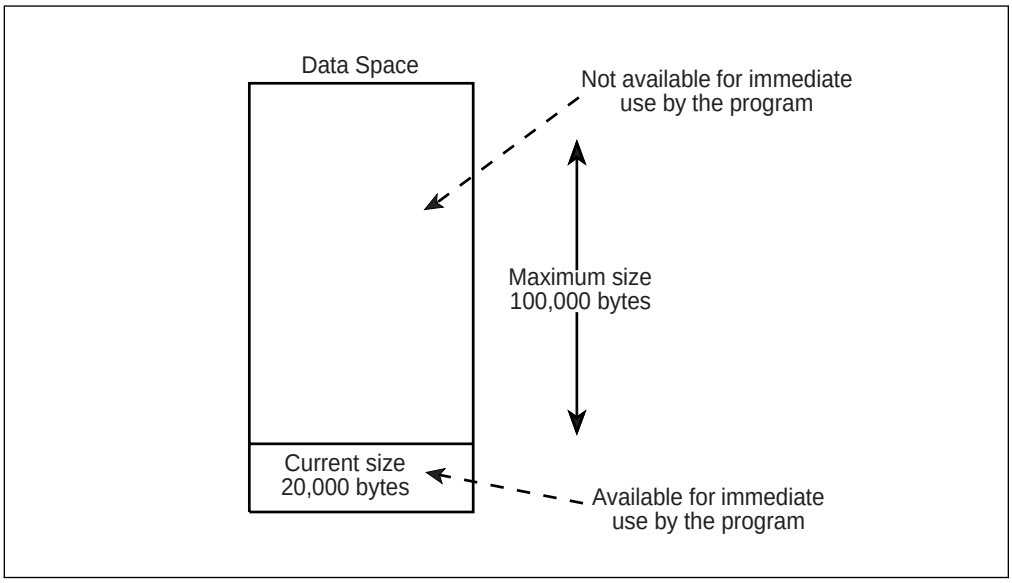


Figure 15-4. Example of Specifying the Size of a Data Space

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 15-14 describes extending the current size and includes an example of how to extend the current size of the data space in Figure 15-4.

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 14, Using Access Registers. “The ALESERV Macro” on page 14-9 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 15-4 on page 15-11 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 15-5 on page 15-16:

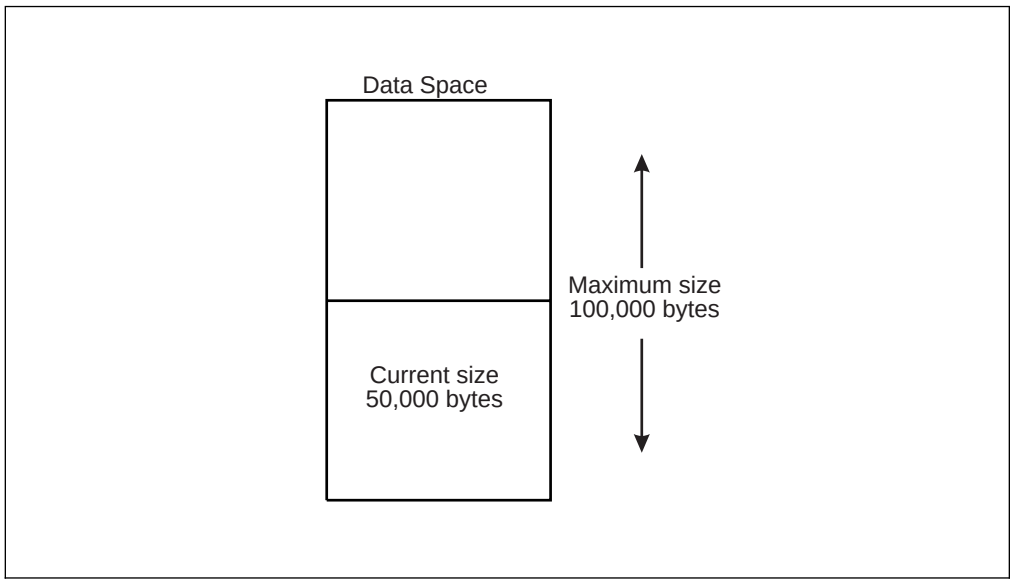


Figure 15-5. Example of Extending the Current Size of a Data Space

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.)

Any task in an address space can page areas into (and out of) central storage from (or to) a data space created by any other task in that address space. Therefore, you can attach a subtask that can preload pages from a data space into central storage for use by another subtask.

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 does not 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 to free resources as soon as they are no longer needed, and to avoid excess processing at termination time. However, if you do not delete the data space, 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 12, 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 15-6 on page 15-19 describes the areas you have defined in the data space.

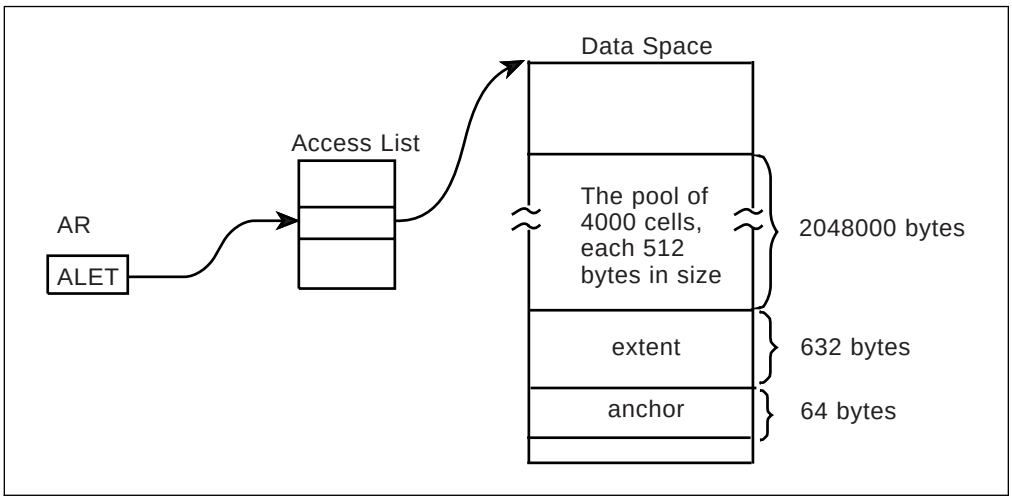


Figure 15-6. Example of Using Callable Cell Pool Services for Data Spaces

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.

A program can use the ETXR option on ATTACH or ATTACHX to specify the address of an end-of-task routine to be given control after the new task is normally or abnormally terminated. The exit routine receives control when the originating task becomes active after the subtask is terminated. The routine runs asynchronously under the originating task. Upon entry, the routine has an empty dispatchable unit access list (DU-AL). To establish addressability to a data space created by the originating task and shared with the terminating subtask, the routine can use the ALESERV macro with the ADD parameter, and specify the STOKEN of the data space.

In the following example, shown in Figure 15-7, 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.

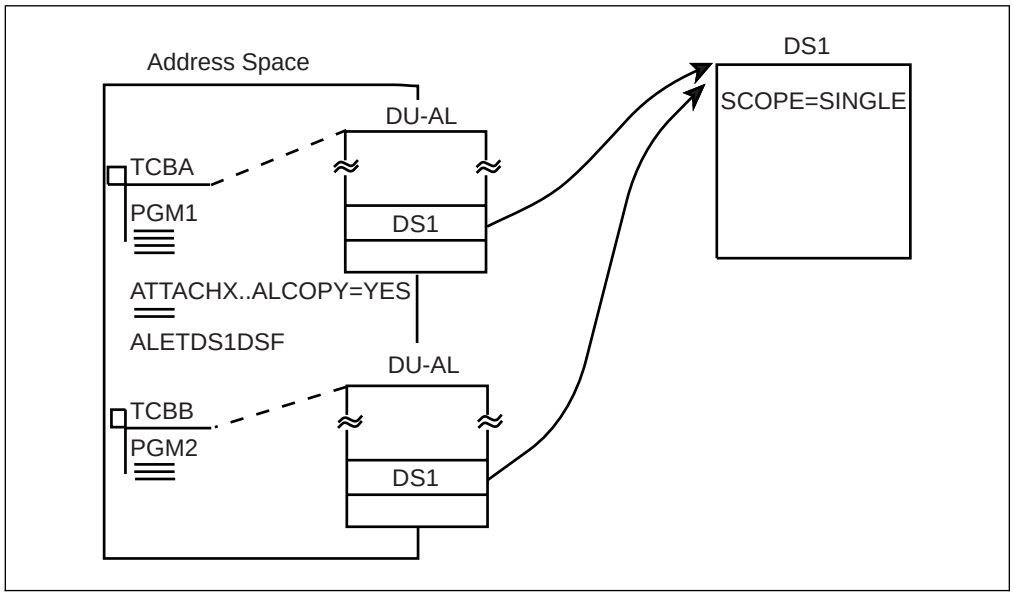


Figure 15-7. Two Problem Programs Sharing a SCOPE=SINGLE Data Space

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

On the SNAPX macro, use the DSPSTOR parameter to dump storage from any addressable data space that the caller can access.

For the syntax of SNAPX, see *OS/390 MVS Programming: Assembler Services Reference*.

Using Checkpoint/Restart

A program can use checkpoint/restart while it has one or more entries for a data space on its access list (DU-AL or PASN-AL). If the program has specified on the ALESERV macro that the system is to ignore entries made to the access list for the data space for checkpoint/restart processing (CHKPT=IGNORE), the CHKPT macro processes successfully.

A program that specifies CHKPT=IGNORE assumes full responsibility for managing the data space storage. Managing the data space storage includes the following:

- If any program depends on the contents of the data space and the data cannot be recreated or obtained elsewhere, the responsible program must save the contents of the data space prior to the checkpoint operation.
- Once the checkpoint operation has completed, the responsible program must perform the following during restart processing to successfully manage the data space storage.
  1. Ensure that the data space exists. The original data space might or might not exist. If the original data space does not exist, the responsible program must issue DSPSERV CREATE to recreate the data space.
  2. Issue ALESERV ADD of the data space, original or recreated, to the program's access list to obtain a new ALET.
  3. If, in addition to having a dependency on the data space, any program also depends on the contents of the data space storage, the responsible program must refresh the contents of the data space storage. The program must use the new ALET to reference the data space.
  4. The responsible program must make the new ALET available to any program that has a dependency on the data space. The STOKEN, changed or unchanged, must be made available to any program that needs to issue ALESERV ADD to access the data space.

See *DFSMS/MVS Checkpoint/Restart* information about the CHKPT macro.

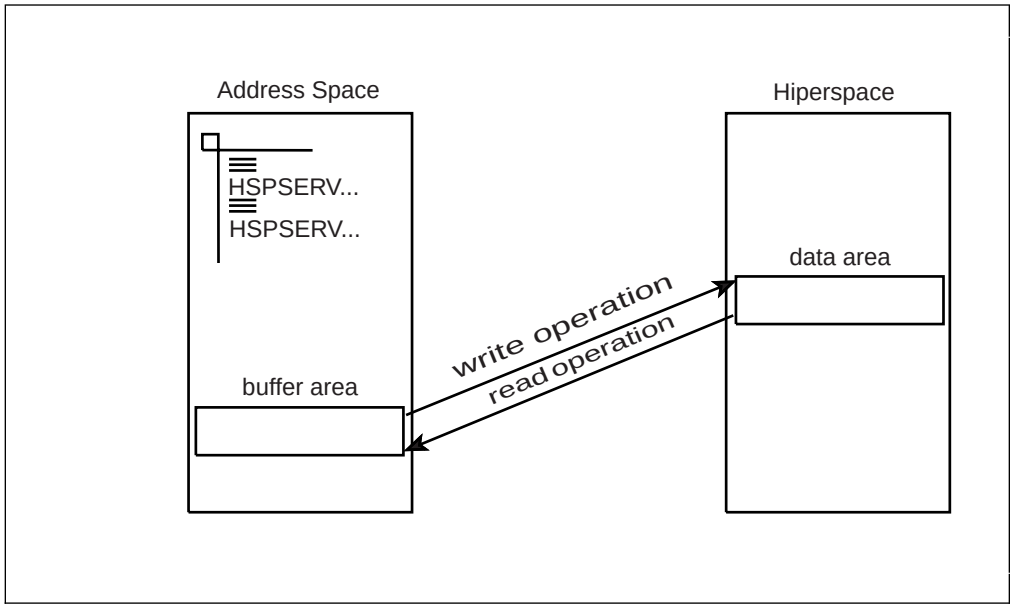
Creating and Using Hiperspaces

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, *OS/390 MVS Programming: Assembler Services 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 15-8 on page 15-27 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.

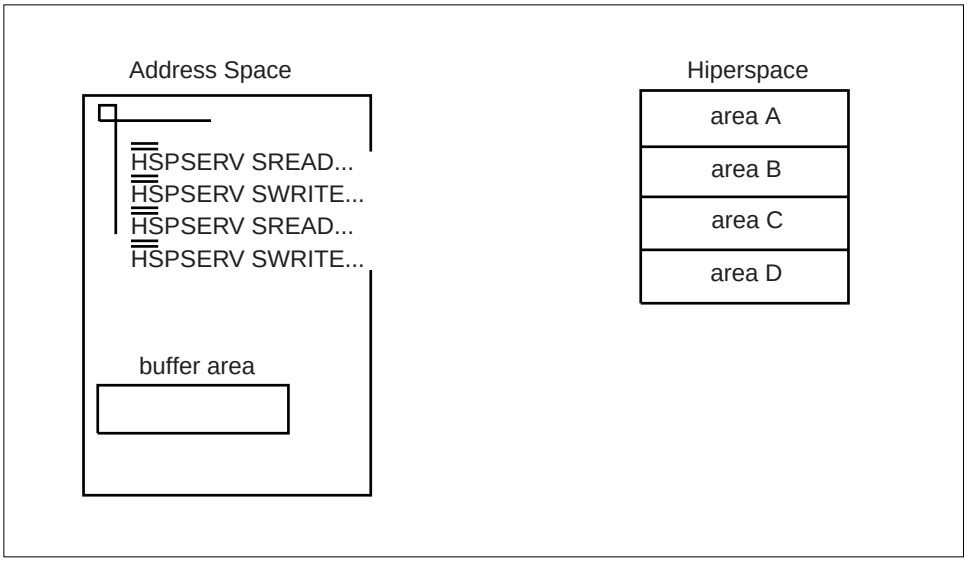


Figure 15-8. Example of Scrolling through a Standard Hiperspace

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 15-37.

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 hiperspace 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 15-9 shows some important facts about non-shared standard hiperspaces:

Figure 15-9. 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 15-9.

- 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 15-10.
- 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 15-11.

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 15-10 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.

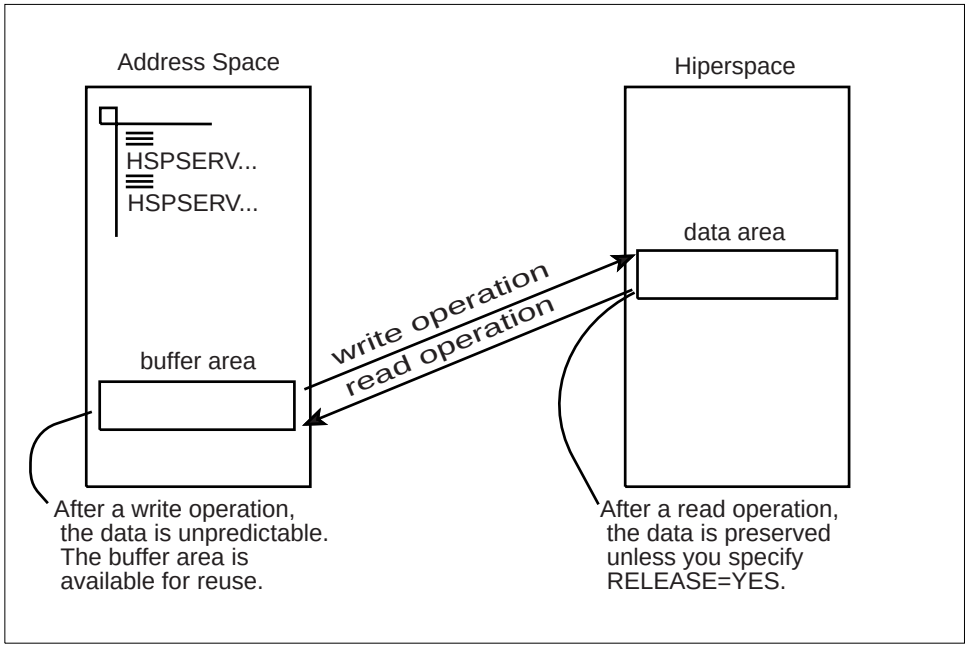


Figure 15-10. Illustration of the HSPSERV Write and Read Operations

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 *OS/390 MVS Programming: Assembler Services 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 15-10 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 15-35 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.

Note: When the HSPALET parameter is specified, the application's RANGLIST data may be modified by the system.

Do not specify RELEASE=YES with the HSPALET parameter.

Programming Notes for Obtaining ALETs for Hiperspaces:

- Using 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 15-37.
- 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 15-19.

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'8192'                * 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 15-14.

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 to free resources as soon as they are no longer needed, and to avoid excess processing at termination time. However, if you do not delete the hiperspace, 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 15-11 on page 15-36 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.

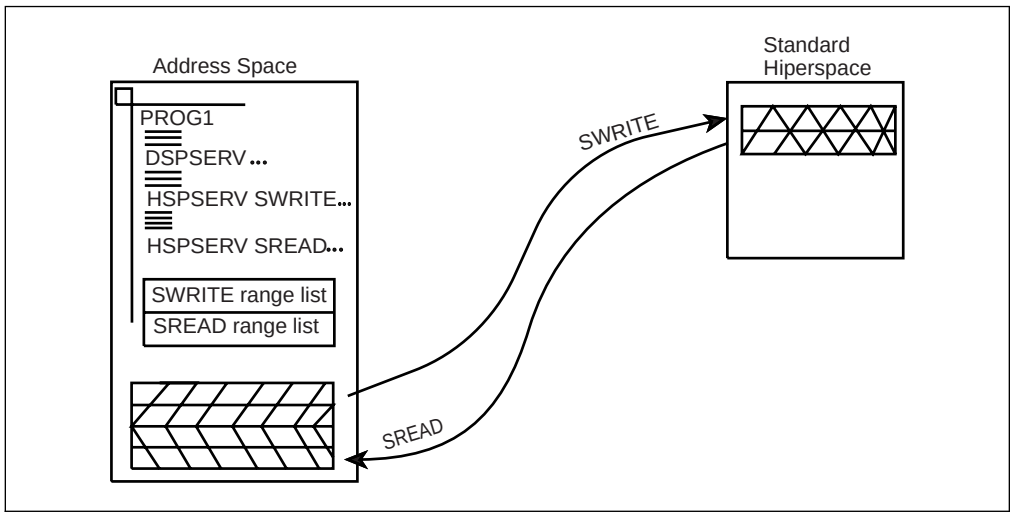


Figure 15-11. Example of Creating a Standard Hiperspace and Transferring Data

```
* 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:

- You can map a VSAM linear data set to hiperspace virtual storage.
- You can map a non-shared hiperspace to virtual storage in an address 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 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 15-31.

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 15-12 shows the hiperspace with a window that begins at the origin of the hiperspace.

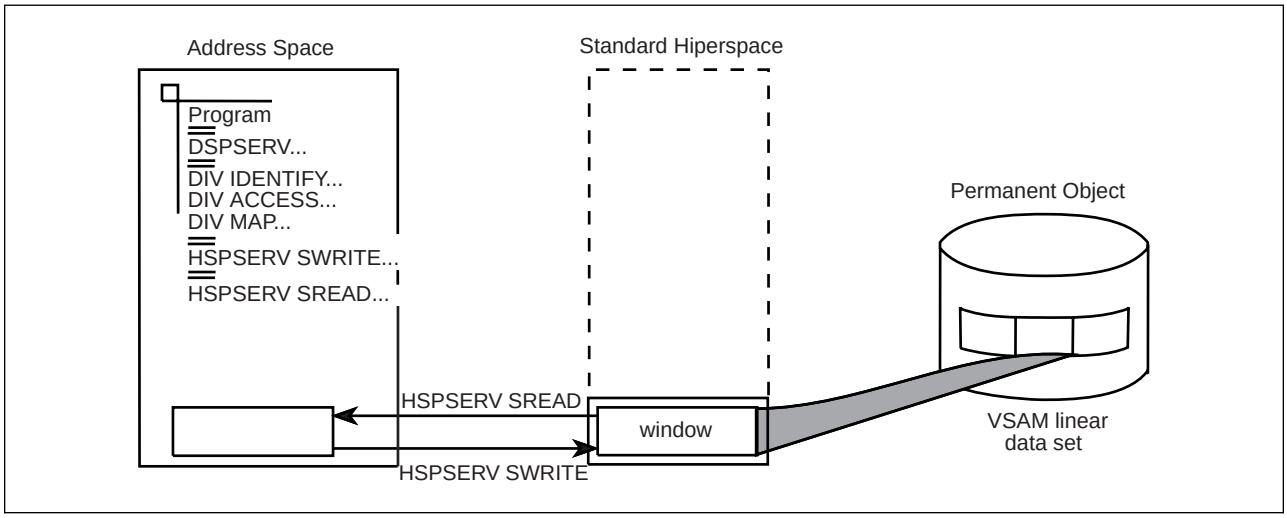


Figure 15-12. Example of Mapping a Data-in-Virtual Object to a Hiperspace

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 15-35.

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 15-13 on page 15-40 shows an example of a hiperspace as a data-in-virtual object.

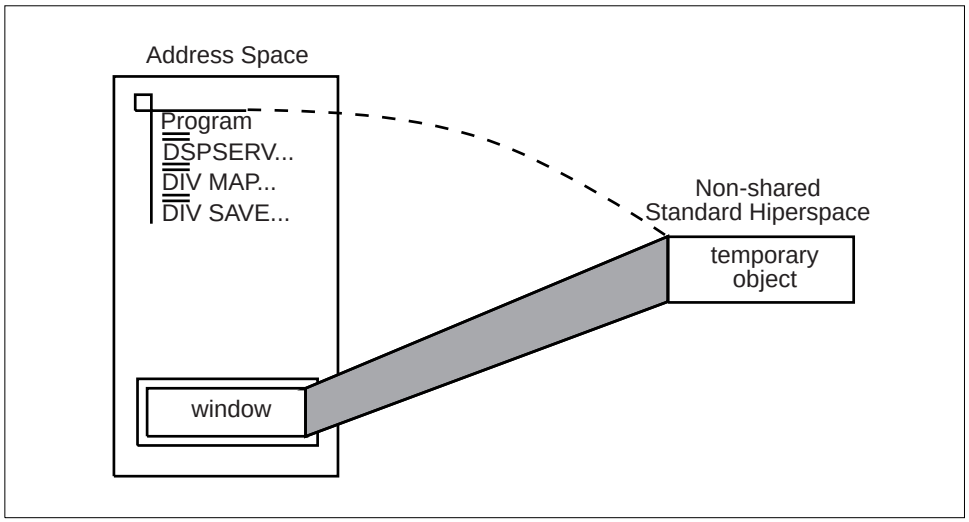


Figure 15-13. A Standard Hiperspace as a Data-in-Virtual Object

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
```

Using Checkpoint/Restart

A program can use checkpoint/restart while it has one or more entries for a hiperspace on its access list (DU-AL or PASN-AL). If the program has specified on the ALESERV macro that the system is to ignore entries made to the access list for the hiperspace for checkpoint/restart processing (CHKPT=IGNORE), the CHKPT macro processes successfully.

A program that specifies CHKPT=IGNORE assumes full responsibility for managing the hiperspace storage. Managing the hiperspace storage includes the following:

- If any program depends on the contents of the hiperspace and the data cannot be recreated or obtained elsewhere, the responsible program must save the contents of the hiperspace prior to the checkpoint operation.
- Once the checkpoint operation has completed, the responsible program must perform the following during restart processing to successfully manage the hiperspace storage.

  1. Ensure that the hiperspace exists. The original hiperspace might or might not exist. If the original hiperspace does not exist, the responsible program must issue DSPSERV CREATE TYPE=HIPERSPACE to recreate the hiperspace.
  2. Issue ALESERV ADD of the hiperspace, original or recreated, to the program's access list to obtain a new ALET.
  3. If, in addition to having a dependency on the hiperspace, any program also depends on the contents of the hiperspace storage, the responsible program must refresh the contents of the hiperspace storage. The program must use the new ALET to reference the hiperspace.
  4. The responsible program must make the new ALET available to any program that has a dependency on the hiperspace. The STOKEN, changed or unchanged, must be made available to any program that needs to issue ALESERV ADD to access the hiperspace.

See *DFSMS/MVS Checkpoint/Restart* information about the CHKPT macro.

Chapter 16. 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.

Data Objects

Permanent

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

© Copyright IBM Corp. 1988, 1999

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.

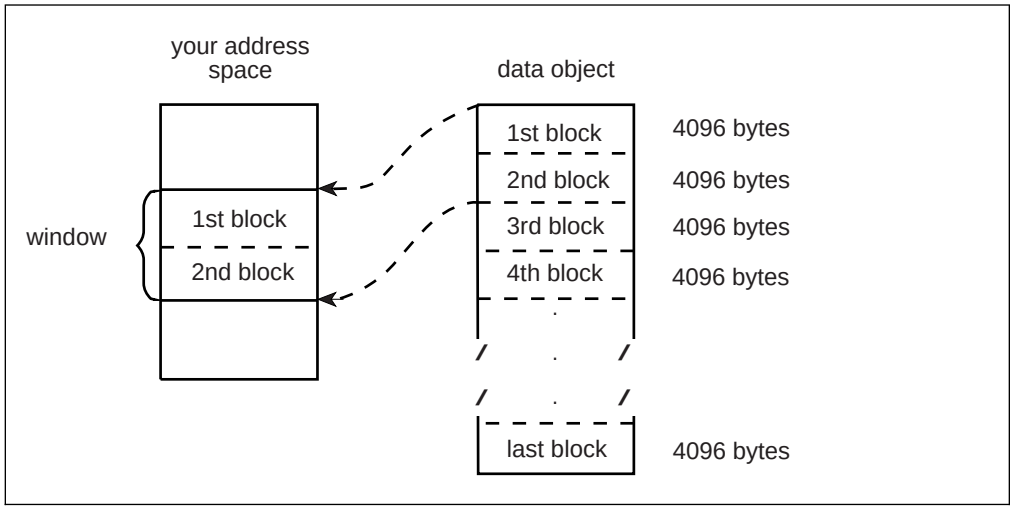


Figure 16-1. Structure of a Data Object

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.

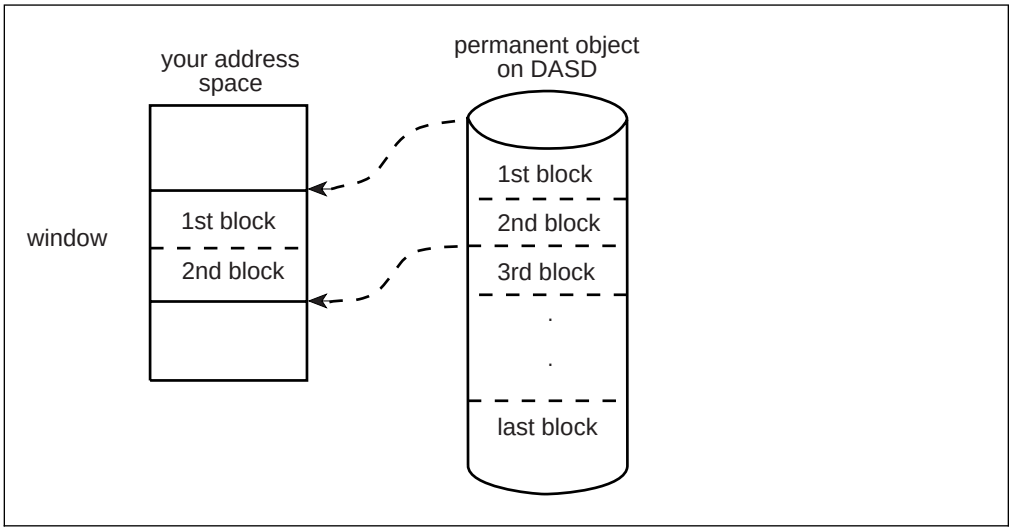


Figure 16-2. Mapping a Permanent Object That Has No Scroll Area

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.

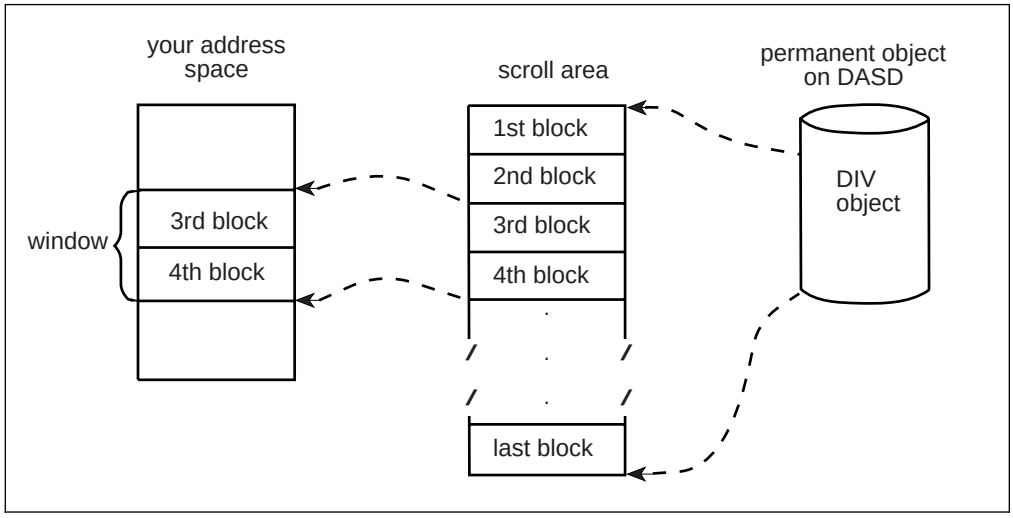


Figure 16-3. Mapping a Permanent Object That Has A Scroll Area

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.

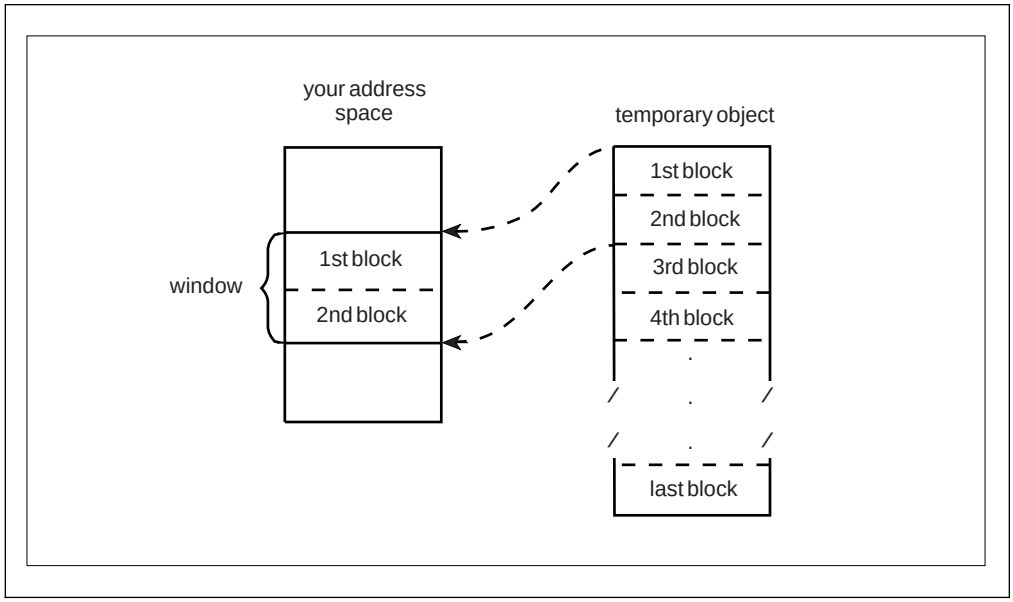


Figure 16-4. Mapping a Temporary Object

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.

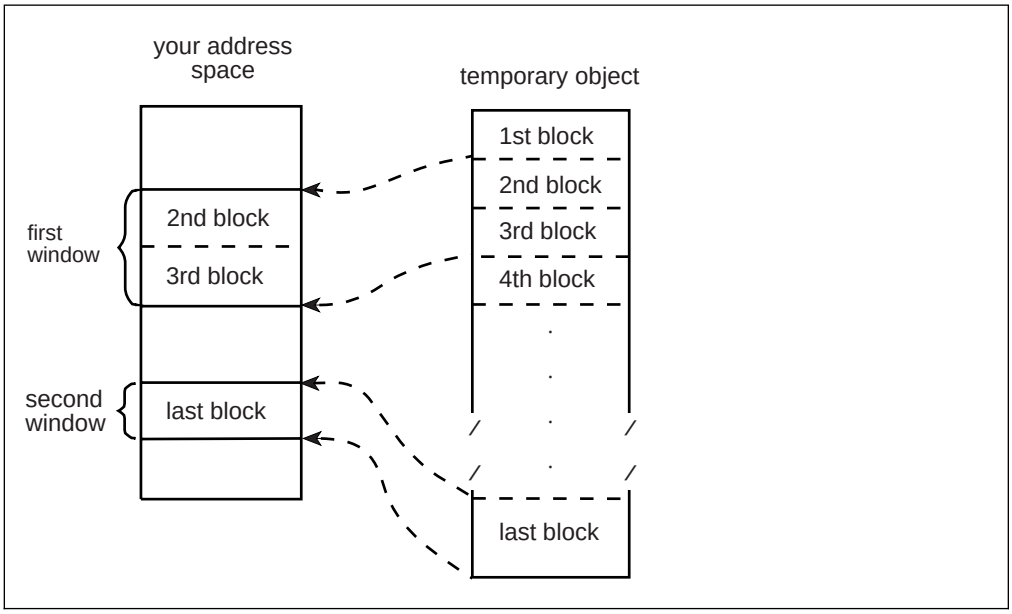


Figure 16-5. Mapping an Object To Multiple Windows

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.

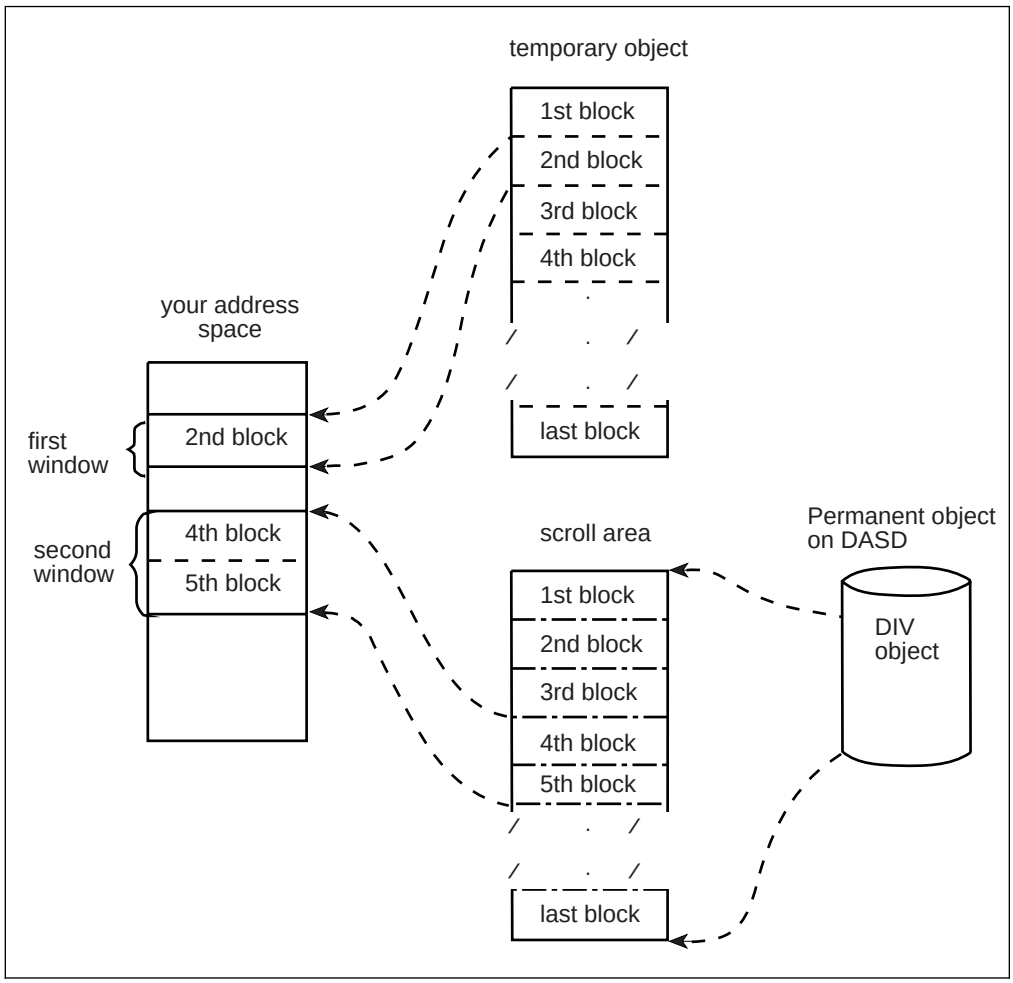


Figure 16-6. Mapping Multiple Objects

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 *OS/390 MVS Programming: Assembler Services Reference*. For an example of invoking window services from an assembler language program, see "Window Services Coding Example" on page 16-20.

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.

The following restrictions apply to using window services:

1. 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 ATTACH or ATTACHX keywords GSPV and GSPL to pass the mapped virtual storage.
2. While your program is in cross-memory mode, your program cannot invoke data-in-virtual services; however, your program can reference and update data in a mapped virtual storage window.
3. The task that obtains the ID (through DIV IDENTIFY) is the only one that can issue other DIV services for that ID.
4. When you identify a data-in-virtual object using the IDENTIFY service, you cannot request a checkpoint until you invoke the corresponding UNIDENTIFY service.

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 SMS, 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 16-12. 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 16-14.

Defining the Disposition of a Window's Contents

You must specify whether window services 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 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=HEWL,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)
  LIBRARY    OBJDD2(CSRIDAC,CSRREFR,CSREVW,CSRSCOT,CSRSAVE,CSRVIEW)
  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....
*        .
*********************************************************************
*  Get 50 pages of virtual storage to use as a window               *
*********************************************************************
         STORAGE OBTAIN,LENGTH=GSIZE,BNDRY=PAGE,ADDR=WINDWPTR
         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 virtual   *
*  storage obtained through the STORAGE macro 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,(R3),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....
*        .
         L  13,SAVEAREA+4
         LM 14,12,12(13)
         BR 14                      End of EXAMPLE1
ZERO     EQU  0                     Constant zero
GSIZE    EQU  204800                Storage for window of 50 pages (blocks)
R3       EQU  3                     Register 3
R4       EQU  4                     Register 4
         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 window's storage
WINDOW   DSECT                    Mapping of window to view the
         DS 204800C               object data
         END
```

Chapter 17. Sharing Application Data (Name/Token Callable Services)

Name/token callable services allow you to share data between two programs running under the same task, or between two or more tasks or address spaces. To share data, programs often need to locate data or data structures acquired and built by other programs. These data structures and the programs using them need not reside in the same address space. Name/token callable services provide a way for programs to save and retrieve the information needed to locate this data.

Both unauthorized (problem state and PSW key 8-15) and authorized programs (supervisor state or PSW key 0-7) can use name/token callable services. Name/token callable services provide additional function that is available to authorized programs only. For a description of those functions, see *OS/390 MVS Programming: Authorized Assembler Services Guide*.

Understanding Name/Token Pairs and Levels

Name/token callable services enable programs to save and retrieve 16 bytes of application-related data. A program can associate a 16-byte character string (the name) with 16 bytes of user data (the token). Later, the same or a different program can retrieve the token by using the name and calling a name/token service.

By using the appropriate name/token callable service, a program can:

- Create a name/token pair (IEANTCR)
- Retrieve a token from a name/token pair (IEANTRT)
- Delete a name/token pair (IEANTDL).

Name/Token Pairs

A name/token pair consists of a 16-byte character string (name) with 16 bytes of user data (token). One program creates the name/token pair, assigns the name, and initializes the token field. Typically, the token is an address of a data structure.

Figure 17-1 shows the name/token pair and indicates its intended use.

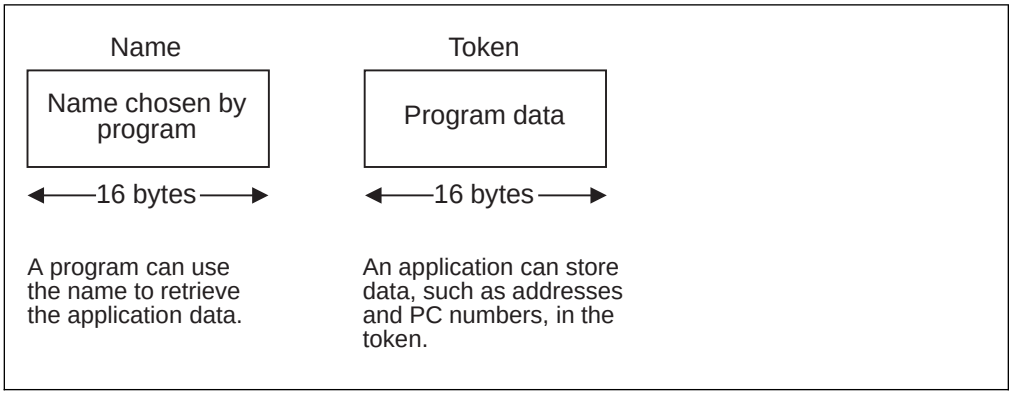


Figure 17-1. Using the Name and the Token

© Copyright IBM Corp. 1988, 1999

The bytes of the name can have any hexadecimal value and consist of alphabetic or numeric characters. The name may contain blanks, integers, or addresses.

Names must be unique within a level. Here are some examples.

- Two task-level name/token pairs owned by the same task cannot have the same name. However, two task-level name/token pairs owned by different tasks can have the same name.
- Two home-address-space-level name/token pairs in the same address space cannot have the same name. However, two home-address-space-level name/token pairs in different address spaces can have the same name.

Because of these unique requirements you must avoid using the same names that IBM uses for name/token pairs. Do not use the following names:

- Names that begin with A through I
- Names that begin with X'00'.

The token can have any value.

Levels for Name/Token Pairs

Name/token pairs have a level attribute associated with them. The level defines the relationship of the creating program (that is, the program that creates the name/token pair) to the retrieving program (that is, the program that retrieves the data). Depending on the level you select, the retrieving program can run under the same task as the creating program, or in the same home address space, in the same primary address space, or in the same system.

- A **task-level name/token pair** allows the creating program and retrieving program to run under the same task.
- A **home-address-space-level name/token pair** allows the creating program and the retrieving program to run in the same home address space.
- A **primary-address-space-level name/token pair** allows the creating program and the retrieving program to run in the same primary address space.
- A **system-level name/token pair** allows the creating program and the retrieving program to run in the same system. That is, the two programs run in separate address spaces.

The various name/token levels allow for sharing data between programs that run under a single task, between programs that run within an address space, and between programs that run in different address spaces. Some examples of using name/token levels are:

- Different programs that run under the same task can share data through the use of a task-level pair.
- Any number of tasks that run within an address space can share data through the use of an address-space pair.

Determining What Your Program Can Do with Name/Token Pairs

The following table shows the name/token callable services your program can use to manipulate different levels of name/token pairs:

Figure 17-2. Summary of What Programs Do with Name/Token Pairs

Service	Level of pairs
Create (IEANTCR)	• Task • Home • Primary
Retrieve (IEANTRT)	• Task • Home • Primary • System
Delete (IEANTDL)	• Task • Home • Primary **Note:** Unauthorized programs cannot delete any pairs created by authorized programs.

Note: The primary-address-space-level and system-level name/token pairs are intended to be used in a cross-memory environment established by authorized programs. For this reason, this book does not contain complete descriptions of these pairs. For complete descriptions of the primary-level and system-level name/token pairs, see *OS/390 MVS Programming: Authorized Assembler Services Guide*.

Deciding What Name/Token Level You Need

To determine the level to use, consider the relationship between the code that creates the pair and the code that retrieves the pair:

- If the retrieving code will be running under the same task as the creator's code, use the task level
- If the retrieving code will have the same home address space but run under a different task, use the home address space level.

Task-Level Name/Token Pair

A task-level name/token pair can be used to anchor data that relates to only one task. Your application program can create and retrieve the data as often as needed.

Figure 17-3 shows the task-level name/token pair for TASK 1.

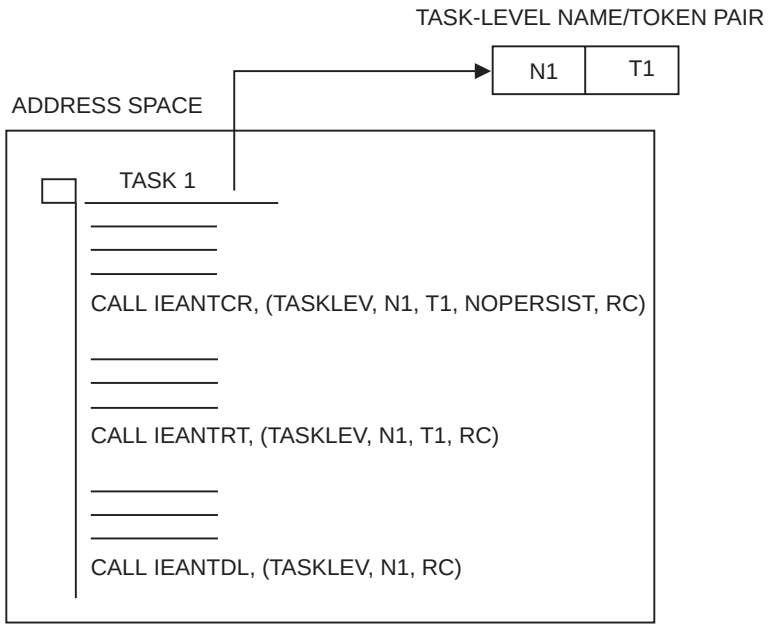


Figure 17-3. Using the Task Level in a Single Address Space

In a single address space, TASK 1:

1. Creates the task-level name/token pair (N1,T1) using the IEANTCR callable service.
2. Retrieves the token at a later time by calling its name (N1) using the IEANTRT callable service.
3. Deletes the name/token pair by calling its name (N1) using the IEANTDL callable service.

Home-Level Name/Token Pair

A home-level name/token pair can anchor data for use by programs running in the creating program's home address space.

Figure 17-4 shows the name/token pairs associated with TASK 1 and TASK 2 running in the address space.

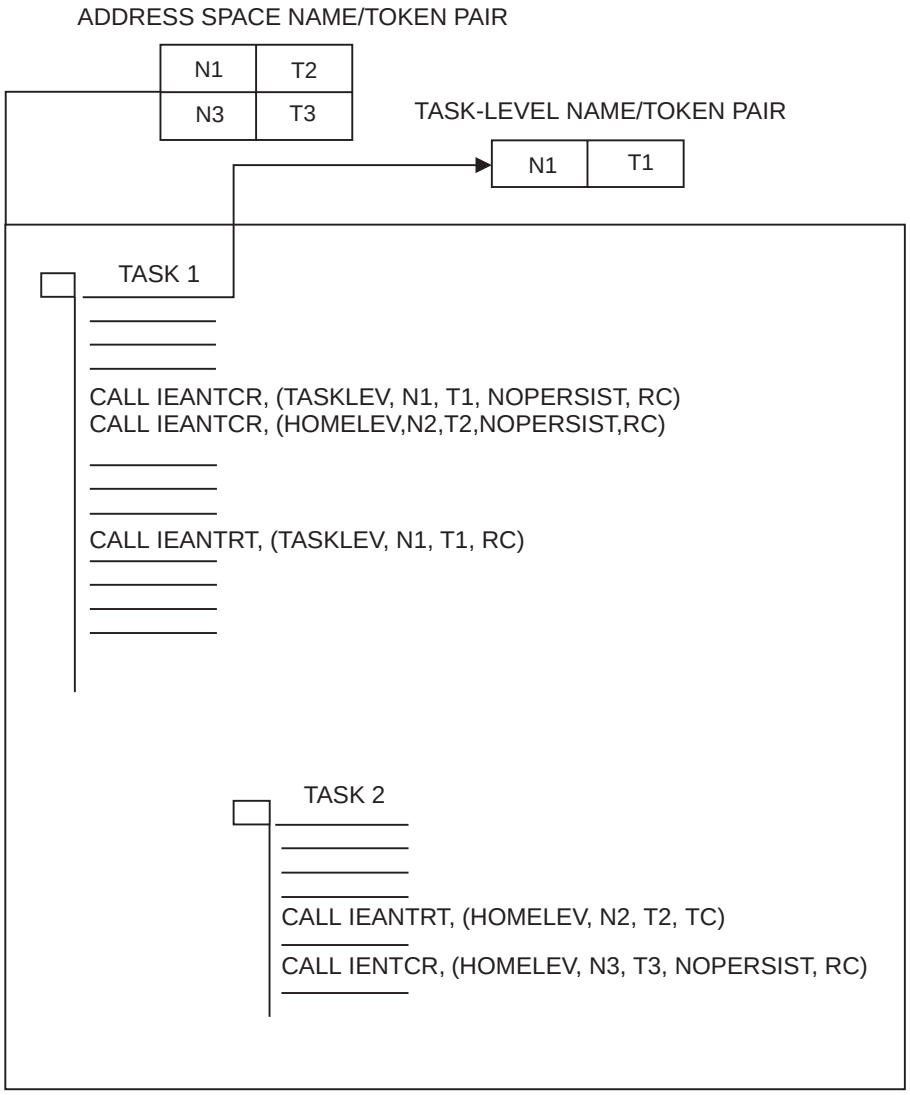


Figure 17-4. Using Home-Level and Task-Level Name/Token Pairs

In Figure 17-4, TASK 1:

1. Creates a task-level name/token pair (N1,T1) and a home-level name/token pair (N2,T2) using the IEANTCR callable service. The home-level name/token pair is associated with the address space.
2. Retrieves the token from N1,T1 any time during the task's processing.

TASK 2 does the following:

1. Retrieves the home-level token from N2,T2 that was created by TASK 1. TASK 2 can retrieve that token because both tasks are running in the same home address space.
2. Creates its own home-level name/token pair (N3,T3) that other tasks running in the home address space can access.

Owning and Deleting Name/Token Pairs

Name/token pairs created by a program are automatically deleted by the system. The level of the pair determines **when** the system deletes the pair:

Note: The words job step in this topic refers to the cross memory resource owning (CRMO) task. While the CRMO task is generally the main job step task, at times it may be either the initiator task or started task control task (such as between jobs).

- Task-level pairs are owned by the task that created them and are deleted when the owning task terminates.
- Home-address-space-level name/token pairs are owned by the job step task of the home address space that created them and are deleted when the job step task, rather than the creating task, in the address space terminates; that is, home-level pairs created by subtasks of a task are not automatically deleted when the subtask terminates.
- Primary-address-space-level name/token pairs are owned by the job step task of the primary address space that created them and are deleted when the job step task, rather than the creating task, in the address space terminates; that is, primary-level pairs created by subtasks of a task are not automatically deleted when the subtask terminates.

Using Checkpoint/Restart with Name/Token Pairs

A program cannot complete a checkpoint operation, by issuing the CHKPT macro, if certain name/token pairs are owned by the program at the time of the checkpoint. The pairs are:

- Task-level name/token pairs owned by the task issuing the CHKPT macro.
- Home- or primary-address-space-level name/token pairs owned by the job step task of the home address space of the task issuing the CHKPT macro.

A checkpoint fails if task- or address-space-level pairs exist because the information in the pairs is not saved in the checkpoint data set when the checkpoint is taken. Because the pair information is not saved, the program cannot be restored when the restart occurs. For more information about checkpoint/restart, see *DFSMS/MVS Checkpoint/Restart*.

Link-Editing Name/Token Services

A program that calls the name/token services must be link-edited with IBM-provided name/token linkage-assist routines. The linkage-assist routines reside in SYS1.CSSLIB. The following example shows the JCL that can link-edit a reentrant program with the linkage-assist routines:

```
//userid   JOB 'accounting-info','name',CLASS=x,
//  MSGCLASS=x,NOTIFY=userid,MSGLEVEL=(1,1),REGION=4096K
//LINKSTEP EXEC PGM=HEWL,
//  PARM='LIST,LET,XREF,REFR,RENT,SIZE=(1800K,128K)'
//SYSPRINT DD   SYSOUT=x
//SYSLMOD  DD   DSN=userid.LOADLIB,DISP=OLD
//SYSLIB   DD   DSN=SYS1.CSSLIB,DISP=SHR
//OBJLIB   DD   DSN=userid.OBJLIB,DISP=SHR
//SYSUT1   DD   UNIT=SYSDA,SPACE=(TRK,(5,2))
//SYSLIN   DD   *
  INCLUDE OBJLIB(userpgm)
  ENTRY userpgm
  NAME userpgm(R)
/*
```

Chapter 18. 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 a range of virtual storage pages be made read-only or be made modifiable.
  - PGSER PROTECT allows the caller to request that a range of virtual storage pages be made read-only.
  - PGSER UNPROTECT allows the caller to request that a range of virtual storage pages be made modifiable.
- Request that the system preload or page out central storage

© Copyright IBM Corp. 1988, 1999

  - 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. The page is no longer available to the issuer. FREEMAIN can free a page that has been protected through the PGSER macro with the PROTECT option. 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 18-1, 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.

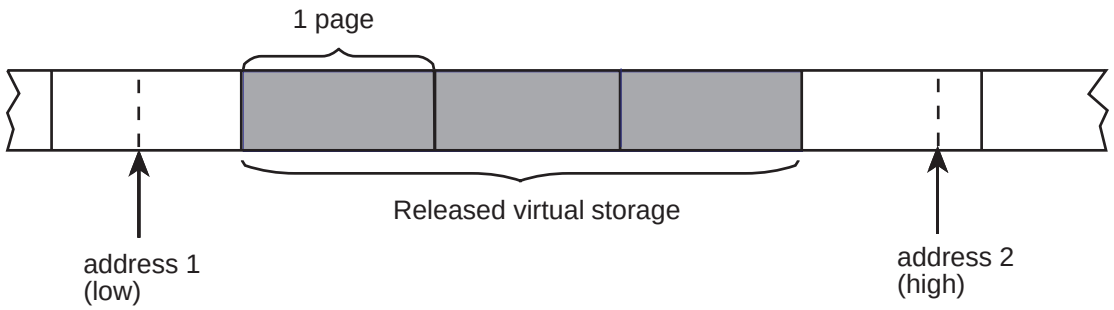


Figure 18-1. Releasing Virtual Storage

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. PGSER RELEASE rejects any attempt to release protected storage, including storage that is dynamically protected through PGSER PROTECT. 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.

Protecting a Range of Virtual Storage Pages

The PROTECT option of PGSER makes a range of virtual storage pages read-only and helps to improve data integrity. The UNPROTECT option of PGSER makes a range of virtual storage pages modifiable. You can protect private storage both above and below 16 megabytes.

IBM recommends that you use PGSER PROTECT only for full pages of storage on page boundaries. This usage avoids making other areas of storage read-only unintentionally. For instance, if you obtain a virtual storage area smaller than a page and then issue PGSER PROTECT, the entire page is made read-only, including the portion of the page that is not part of your storage area.

The system does not keep track of how many times a page has been protected or unprotected. One UNPROTECT cancels all PROTECTs for a given page.

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. 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 15-16.

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 non-pageable and contained in the address space of the subarea to be processed.

Each subarea list entry has the format shown below. The flag bits that are described are the only flag bits intended for customer use.

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.

Start Address:

The 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 3	(...1 ....)	This flag indicates that a return code of 4 was issued from a page service function other than PGRLSE.

End Address + 1:

The 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. 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. 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.
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. You do not need to do anything with bytes 4-7 if you supplied a pointer in bytes 0-3.
8	Flags set by the caller as follows. The flag bits that are described are the only flag bits intended for customer use.

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	For IBM use only.

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 *OS/390 MVS Programming: Callable Services for HLL*.

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 18-2 on page 18-7 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 18-2 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.

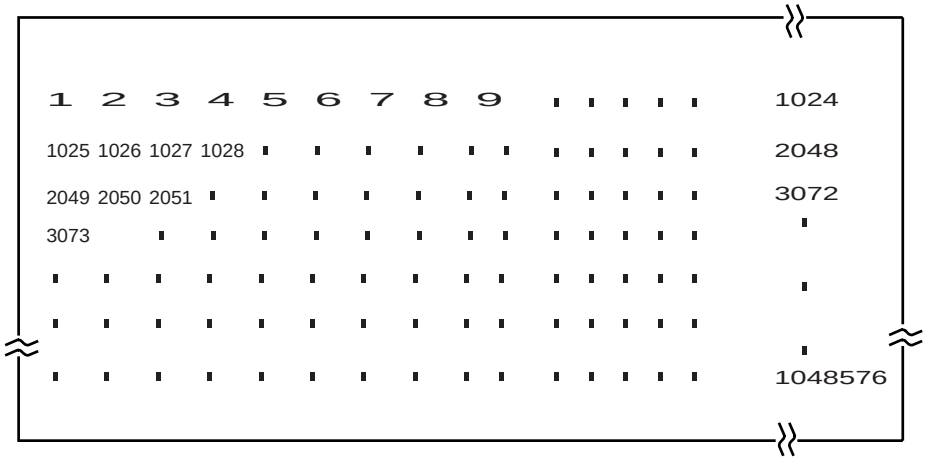
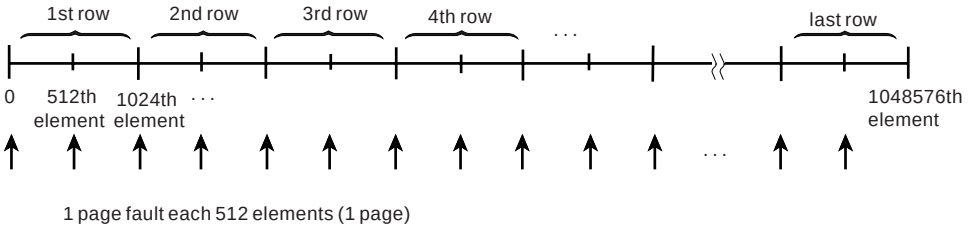


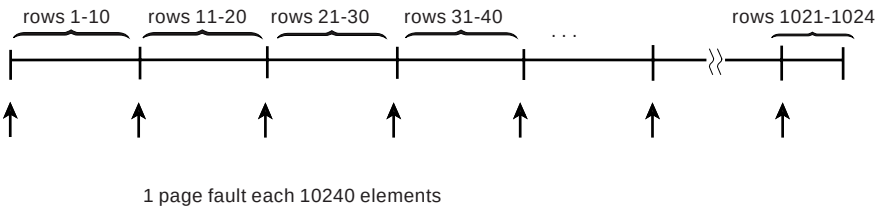
Figure 18-2. Example of using REFPAT with a Large Array

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.



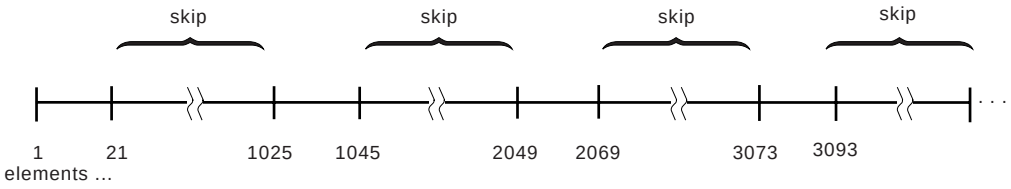
By bringing in one page at a time, the system takes 2048 page faults (8388608÷4096), each page fault adding to the elapsed time of the program.

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 (1004×8) every 8192 bytes (1024×8). 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 18-3 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.

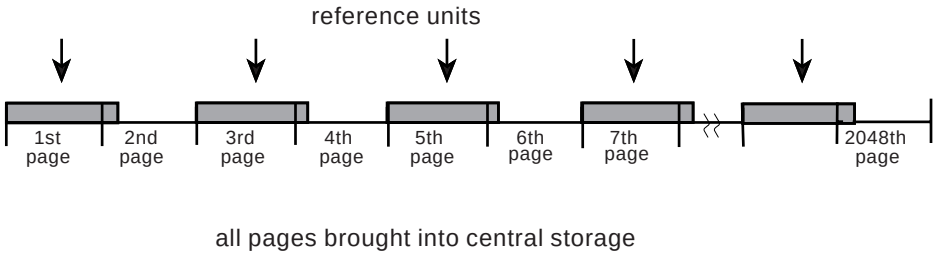
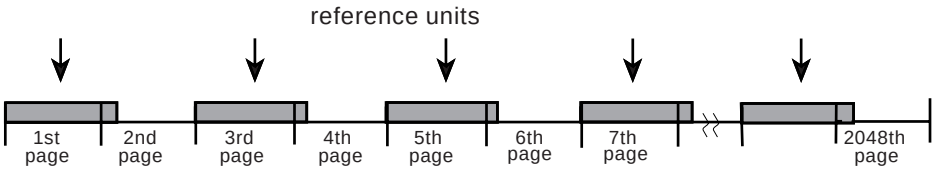


Figure 18-3. Illustration of a Reference Pattern with a Gap

Every other page of the data does not come into central storage; those pages contain only the “skipped over” data.

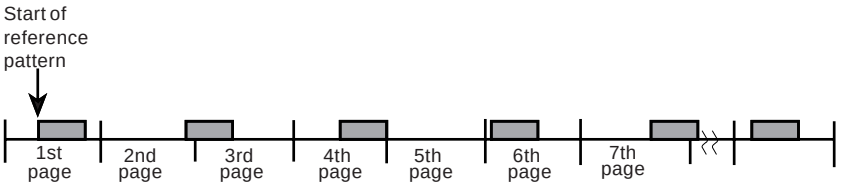
Example 2: The reference pattern includes a reference unit of 4800 bytes and a gap of 3392 bytes. The example assumes that the area to be referenced starts on a page boundary.



all pages brought into central storage

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 18-11 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 18-12 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 18-4 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.

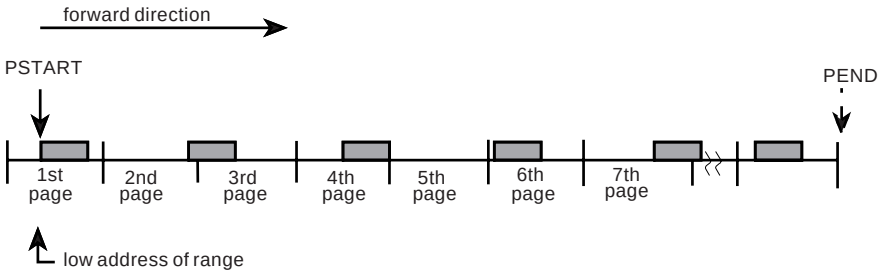


Figure 18-4. Illustration of Forward Direction in a Reference Pattern

Figure 18-5 on page 18-11 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.

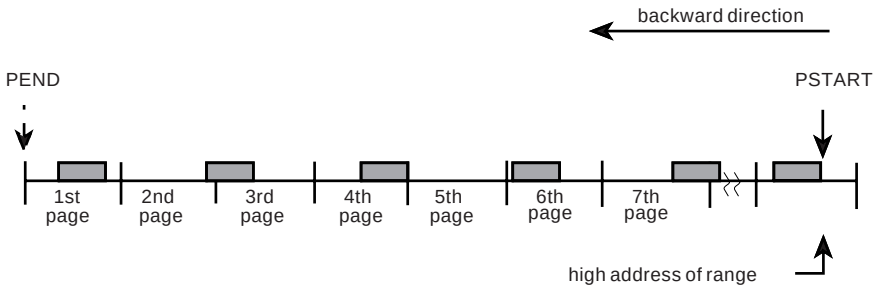


Figure 18-5. Illustration of Backward Direction in a Reference Pattern

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 18-6 identifies two reference patterns that characterize most of the reference patterns that REFPAT applies to. The marks on the line indicate referenced elements.

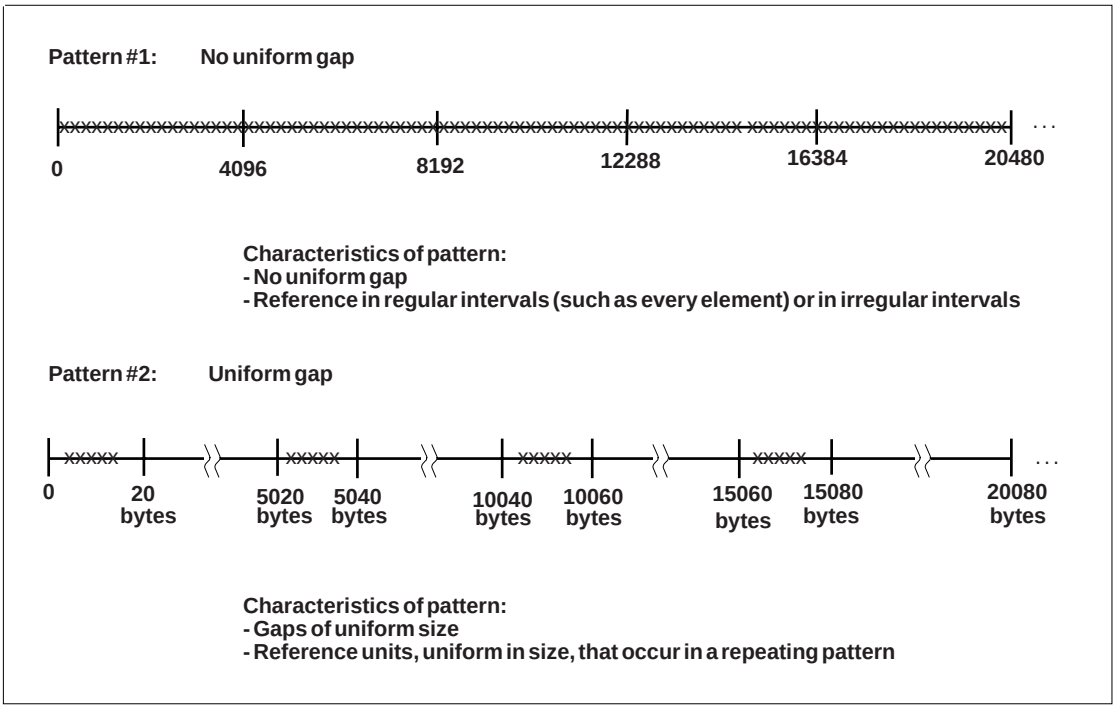


Figure 18-6. Two Typical Reference Patterns

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" 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 18-6 on page 18-11 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 18-8 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. See *OS/390 MVS Initialization and Tuning Guide* for information on tuning the system.

  REFPAT affects movement of pages from auxiliary **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.

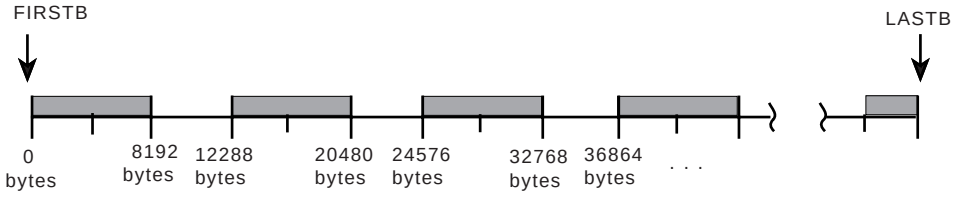
Example 1: The program processes an array in a consistently forward direction from one reference unit to the next. The processing of each element is fairly simple. The program runs during the peak hours and many programs compete for processor time and central storage. A reasonable value to choose for 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 (4×8192) 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 19. Sharing Data in Virtual Storage (IARVSERV Macro)

With the shared pages function, which is available through the IARVSERV macro, you can define virtual storage areas through which data can be shared by programs within or between address spaces or data spaces. Also, the type of storage access can be changed.

Sharing reduces the amount of processor storage required and the I/O necessary to support data applications that require access to the same data. For example, IARVSERV provides a way for a program running below 16 megabytes, in 24-bit addressing mode, to access data above 16 megabytes that it shares with 31-bit mode programs. IARVSERV allows the sharing of data without the central storage constraints and processor overhead of other existing methods of sharing data.

The sharing of data benefits many types of applications, because data is available to all sharing applications with no increase in storage usage. This function is useful for applications in either a sysplex environment or a single-system environment. Additionally, IARVSERV allows you to control whether a sharing program:

- Has read access only
- Has both read and write access and receives updates immediately
- Can modify the data without modifying the original, and without allowing the sharing programs to view the updates
- Can modify the original while sharing programs see the change, but without allowing the sharing programs to change the data
- Can change the current storage access

An additional macro, IARR2V, is provided as an aid to converting central storage addresses to virtual storage addresses. See “Converting a Central to Virtual Storage Address (IARR2V Macro)” on page 19-9 for information on the IARR2V macro.

The IARVSERV topics described in this chapter are:

- Understanding the concepts of sharing data with IARVSERV
- Storage you can use with IARVSERV
- Obtaining storage for the source and target
- Defining storage for sharing data and access
- Changing storage access
- How to share and unshare data
- Accessing data in a sharing group
- Example of sharing storage with IARVSERV
- Use with data-in-virtual (DIV macro)
- Diagnosing problems with shared data

For coding information about the IARVSERV and IARR2V macros, see *OS/390 MVS Programming: Assembler Services Reference*.

© Copyright IBM Corp. 1988, 1999

Understanding the Concepts of Sharing Data with IARVSERV

As you read this section, refer to Figure 19-1 for an illustration of the sharing data through the IARVSERV macro.

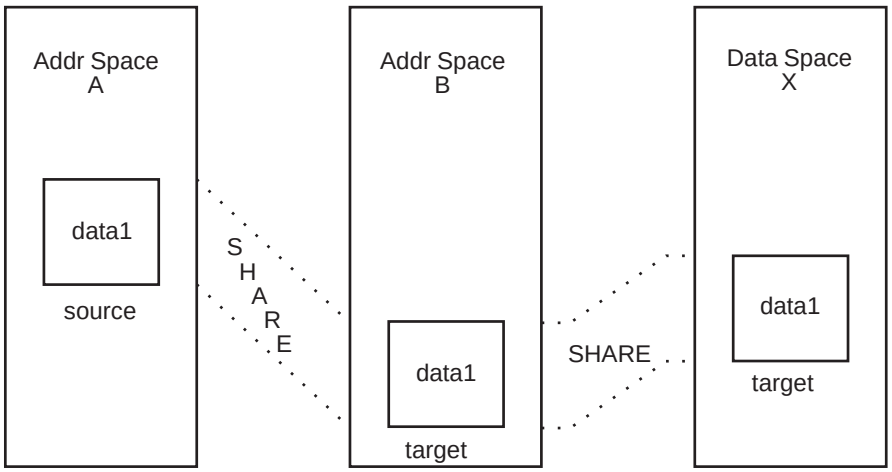


Figure 19-1. Data Sharing with IARVSERV

Suppose that Addr Space A contains data that is required by programs in Addr Space B. A program in Addr Space A can use IARVSERV to define that data to be shared; that data and the storage it resides in are called the **source**. The program also defines storage in Addr Space B to receive a copy of the source; that storage and its copy of source data are called the **target**.

The source and its corresponding target form a **sharing group**. A sharing group can consist of several target areas and one source. For example, suppose another program in Addr Space A defines a portion of data1 (in Addr Space A) as source, and defines a target in Data Space X. That target becomes a member of the sharing group established previously.

All sharing of data is done on a page (4K) basis. If the source page is already a member of an existing sharing group, the target becomes a member of that existing sharing group. A page is called a **sharing page** if it is a member of a sharing group.

Programs that access the source or targets are called **sharing programs**. Each sharing program accesses the shared virtual storage as it would any other storage, and may not need to know that the storage is being shared. So, you can allow programs to share data through IARVSERV without having to rewrite existing programs.

Storage You Can Use with IARVSERV

You can share data in address spaces and data spaces. You can use any storage to which you have valid access, except for a hiperspace, a VIO window, a V=R region, or PSA.

The maximum number of shared pages for a program in problem state with PSW key 8-15 is 32, unless this number is modified by your installation. This number includes both the source and targets, so the actual number of unique pages is 16.

Obtaining Storage for the Source and Target

Before you can issue IARVSERV to define storage as shared, you must obtain or create both the source and target areas. For address space storage, use the GETMAIN or STORAGE macro; for data space storage, use the DSPSERV macro. The source and target areas must be as follows:

- Start on a page boundary,
- Have the same storage protect key and fetch-protection status (except for TARGET_VIEW=UNIQUEWRITE or TARGET_VIEW=LIKESOURCE and the source has UNIQUEWRITE view),
- Meet one of the following requirements:
  - Reside within pageable private storage of an address space.
  - Reside within the valid size of an existing data space and be pageable storage.

The source and the target must be two different storage areas. They must be different virtual storage addresses or reside in different address or data spaces.

Then initialize the source with data. Make sure any storage you obtain or data space you create can be accessed by the intended sharing programs. For example, if you want to allow sharing programs to both read and modify a target, the programs' PSW key value must match or override the target's storage protection key. For information on data spaces, see Chapter 15, “Data Spaces and Hiperspaces” on page 15-1.

Defining Storage for Sharing Data and Access

With the IARVSERV macro, you can define multiple types of data sharing and access. As you read this section, use Figure 19-1 on page 19-2 to see how each IARVSERV parameter acts on the current state of the data. Each type of data sharing access is called a specific **view** of the source data. A view is the way your program accesses, or sees, the data. You define the view in the TARGET_VIEW parameter on IARVSERV, by specifying one of the following:

- Read-only view (READONLY value) — where the target data may not be modified.
- Shared-write view (SHAREDWRITE value) — where the target data can be read and modified through the view.
- Copy-on-write view (UNIQUEWRITE value) — where the source data modifications are not seen by other source - sharing programs. Any attempt to modify the shared source data in this view causes MVS to create a unique target copy of the affected page for that address or data space.

  An example of two different cases:

  - If the shared data is modified through a SHAREDWRITE view, the UNIQUEWRITE view gets an unmodified copy of the data. Any remaining views sharing that data see the modified data.
  - If the shared data is modified through a UNIQUEWRITE view, the UNIQUEWRITE view gets the modified copy of the data. Any remaining views sharing that data see the unmodified data.

- Copy-on-write target view (TARGETWRITE value) — where the target data may be read and modified through the source view. Any modification of a shared target page causes MVS to create a unique target copy of the affected page for that address or data space.

  An example for two different cases:

  – If the shared data is modified through a SHAREDWRITE view, the TARGETWRITE view sees the modified data.

  – If the shared data is modified through a TARGETWRITE view, the TARGETWRITE view sees the modified copy of the data. Any remaining views sharing that data see the unmodified data.

- Like source view (LIKESOURCE value) — where the target data is given the current view type of the source data. If the source data is currently not shared, then its current storage attribute is given to the target.
- Hidden view (HIDDEN value) — where the target will share the source data, but any attempt to access the target data (HIDDEN value) will cause a program check. To access the target, the view type must be changed to READONLY, SHAREDWRITE, UNIQUEWRITE, or TARGETWRITE.

When you specify a value for TARGET_VIEW, keep the following in mind:

- The execution key (PSW key) of the caller must be sufficient for altering the target area. If TARGET_VIEW=SHAREDWRITE is specified, the execution key must be sufficient for altering the source area also.
- For TARGET_VIEW=UNIQUEWRITE, if the input source area is address space storage, and the storage has not been obtained by GETMAIN or STORAGE OBTAIN, or the storage and fetch protect keys do not match, then the SHARE is not performed for that area. The target will be all zeros (first reference), or it will remain as pages that were not obtained by GETMAIN.
- For target views created with LIKESOURCE on IARVSERV SHARE, the system propagates explicit page protection from the source to the target view.
- Page-fixed pages and DREF pages cannot be made TARGETWRITE, UNIQUEWRITE, or HIDDEN.

Changing Storage Access

With the IARVSERV macro, the SHARE and CHANGEACCESS parameters can change the views type of storage access. For SHARE, the current storage attribute of the source data affects the outcome of the target. Figure 19-2 shows the permitted target views for different combinations with the source. A NO in the table means that an abend will occur if you request that target view with the current source view. For CHANGEACCESS, all combinations are permitted.

Figure 19-2. Allowed Source/Target View Combinations for Share

Current	Requested Target View					
Source View	**READONLY**	**SHAREDWRITE**	**UNIQUEWRITE**	**TARGETWRITE**	**HIDDEN**	**LIKESOURCE**
READONLY	Yes	No	Yes	Yes	Yes	Yes
SHAREDWRITE	Yes	Yes	Yes	Yes	Yes	Yes
UNIQUEWRITE	Yes	Yes	Yes	Yes	Yes	Yes
TARGETWRITE	No	No	Yes	No	No	Yes
HIDDEN (Shared)	No	No	No	No	No	Yes
Non-Shared	Yes	Yes	Yes	Yes	Yes	Yes
HIDDEN (Non-Shared)	No	No	No	No	No	Yes

The following apply when using IARVSERV SHARE when changing storage access:

- For source views to be either UNIQUEWRITE or TARGETWRITE, the processor must have the Suppression-On-Protection (SOP) hardware feature, and a previous IARVSERV SHARE must have created a view of UNIQUEWRITE or TARGETWRITE.
- For target views to be TARGETWRITE, the processor must have the SOP hardware feature. If a request is made to create a TARGETWRITE view and the SOP feature is not installed, the request fails with a return code of 8.
- For target views to be UNIQUEWRITE, the SOP hardware feature must be installed. Also, the request must not specify COPYNOW. If the request specifies COPYNOW, or the SOP feature is not installed, a UNIQUEWRITE view is not established, and a separate copy of the data is made.
- For target views created with LIKESOURCE on IARVSERV SHARE, the system propagates explicit page protection from the source to the target view.
- For source pages that are not shared, if the page is page-protected, the view created for that page is a SHAREDWRITE view, but the view is flagged as an explicitly protected view (one that cannot be modified).

The following apply when changing the storage access with IARVSERV CHANGEACCESS:

- To remove hidden status, you must use an IARVSERV CHANGEACCESS, FREEMAIN, or DSPSERV DELETE macro.
- To remove explicit read-only protection status, you must use an IARVSERV CHANGEACCESS, FREEMAIN, DSPSERV DELETE, or PGSER UNPROTECT macro.
- If a hidden page is hidden because of loss of access to 'mapped' data (such as through DIV UNMAP), and, if the page is changed from hidden, the data in the page might be lost.
- Hidden pages cannot be released via a PGSER RELEASE or DSPSERV RELEASE macro. An attempt would result in an abend with the same reason code as is used for protected pages.
- Issuing an IARVSERV UNSHARE macro for the original mapped page causes the data to be retained for that page. The data for the other sharing pages is lost. References to hidden pages cause an X'0C4' abend, and references to lost pages cause in a X'028' abend.

- Page-fixed pages and DREF pages cannot be made TARGETWRITE, UNIQUEWRITE, or HIDDEN.

How to Share and Unshare Data

With the IARVSERV macro, use the SHARE parameter to initiate sharing of data; use the UNSHARE parameter to end sharing for the issuing program. This section discusses the additional IARVSERV parameters that you can specify with SHARE or UNSHARE.

The **RANGLIST** parameter is always required for both SHARE and UNSHARE. It gives IARVSERV information about the source and target addresses. The RANGLIST value is actually the address of the list of addresses you must create using the mapping macro IARVRL. See *OS/390 MVS Data Areas, Vol 2 (DCCB-ITTCTE)* for the details of IARVRL. The following table lists the required IARVRL fields that you must supply for SHARE or UNSHARE.

IARVRL Fields That You Must Initialize for SHARE	IARVRL Fields That You Must Initialize for UNSHARE
VRLSVSA VRLSSTKN (for STOKEN) VRLSALET (for ALET) VRLNUMPG VRLTVSA VRLTSTKN (for STOKEN) VRLTALET (for ALET)	VRLNUMPG VRLTVSA VRLTSTKN (for STOKEN) VRLTAKET (for ALET)

For IARVSERV SHARE, if the target area contains pages that belong to an existing sharing group, MVS performs an implicit UNSHARE to pull those pages out of the existing sharing group before proceeding. Also, MVS automatically performs an UNSHARE on any sharing page when the page is being freed by FREEMAIN, STORAGE RELEASE, or DSPSERV DELETE, or when the page's address space is ended.

Also, when MVS finds that one page of a range is not acceptable for sharing, MVS will not complete the SHARE request for that page, nor the rest of the range or ranges not already processed. You can assume that all pages up to that point were processed successfully. An abend will be issued and GPR 2 and 3 will contain the address range list associated with the error page and the storage address of the page in error, respectively. To remove the SHARE on the successful pages, issue IARVSERV UNSHARE for the storage ranges up to, but excluding, the error page.

The parameter **TARGET_VIEW** is required with SHARE only, to tell IARVSERV how you plan to share the data contained in the source. You have three choices described in “Defining Storage for Sharing Data and Access” on page 19-3.

- READONLY does not allow any program accessing the target area to write to it. An abend results if a program attempts to write to a READONLY target.
- SHAREDWRITE allows any sharing program to write to the target. All those sharing the target area instantly receive the updates. This view could be very useful as a communication method for programs.
- UNIQUEWRITE has the property of copy-on-write, which means that MVS creates a copy of a page for the updating program once the program writes to

that page. The only program that has the change is the program that changed it; all others continue to use the original page unmodified. This is true whether the program writes to a source or target page.

A copy-on-write hardware facility is provided for additional performance improvement. If you need to determine if your processor has the feature, you can use the CVT mapping macro, and test the CVTSOPF bit. See *OS/390 MVS Data Areas, Vol 1 (ABEP-DALT)* for details on the CVT mapping macro.

RETAIN is a parameter available only with UNSHARE. RETAIN=NO requests that MVS remove the target from sharing. The target data is lost. RETAIN=YES requests that MVS leave the data in the target untouched.

Accessing Data in a Sharing Group

Data is accessed in a sharing group just as it would be if sharing did not exist. Trying to write to a READONLY view will cause an abend.

You can create a sharing group that permits programs in 24-bit addressing mode to access data above 16 megabytes. To do this, you would define the source in storage above 16 megabytes, and obtain a target in storage below 16 megabytes. Then initialize the source with data, so programs in 24-bit mode can share the data through the target.

Example of Sharing Storage with IARVSERV

Suppose you are updating a program called PGMA, that controls all the account deposits for a savings bank. Your program must work with two older programs that are complex and do not have source code available. The first program, called SUBPGMA, was updated six years ago and runs in 31-bit addressing mode; it records deposits in money market accounts. It cannot use data spaces. The other program, SUBPGMB, is much older and records deposits in standard savings accounts. It runs in 24-bit addressing mode. See Figure 19-3 on page 19-8 for a representation of the storage.

Program PGMA, the main program, was written to keep all of its data in one large data space. PGMA must continually obtain appropriate storage in the address space that is addressed by SUBPGMA and SUBPGMB. After SUBPGMA and SUBPGMB finish, PGMA must copy all the updated data back to the data space. This is degrading performance and needs to be fixed. By using IARVSERV, you can eliminate the copying, and reduce the complexity of PGMA.

Your update to PGMA would cause the programs to work together this way:

1. PGMA creates a data space and initializes it with data.
2. Before PGMA calls SUBPGMA to do a money market deposit, PGMA issues GETMAIN for storage in the private area for a buffer. This buffer is BUFFER31.
3. PGMA issues IARVSERV SHARE to share the source in the data space with the target, BUFFER31. Use TARGET_VIEW=SHAREDWRITE so updates can be made directly into the data space.
4. PGMA now calls SUBPGMA to update the data, passing the address of BUFFER31 as the area to be updated.

5. Once SUBPGMA updates the data in BUFFER31, PGMA issues IARVSERV UNSHARE followed by FREEMAIN to release the storage.
6. When PGMA needs to call SUBPGMB to do a savings account deposit, the only difference is that PGMA must obtain storage below 16 megabytes for the buffer. This buffer is BUFFER24.
7. PGMA again issues IARVSERV SHARE with TARGET_VIEW=SHAREDWRITE, but identifies the target as BUFFER24.
8. PGMA calls SUBPGMB to update the data, passing the address of BUFFER24 as the area to be updated.
9. Once SUBPGMB updates the data in BUFFER24, PGMA issues IARVSERV UNSHARE and FREEMAIN to release the storage as before.

Note that all three programs could share the data in the data space at the same time. Sharing continues until PGMA issues IARVSERV UNSHARE for that buffer area.

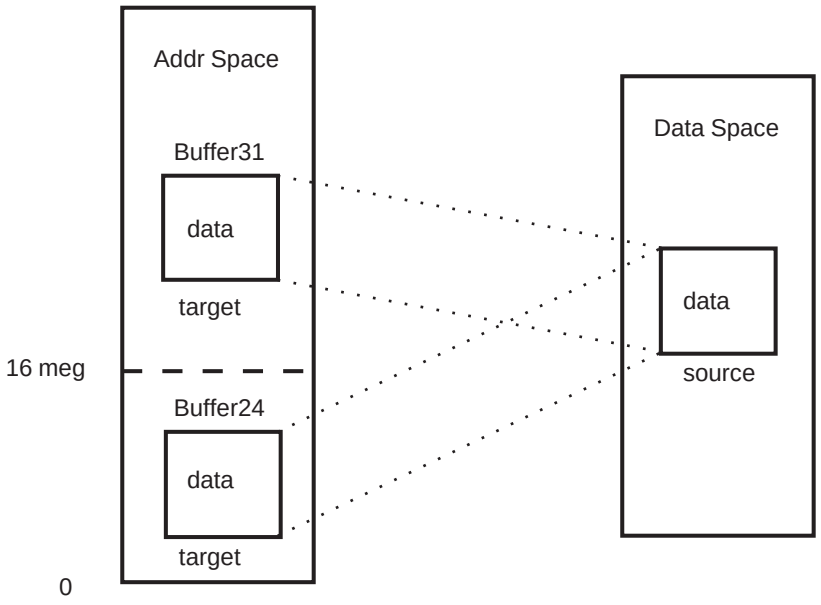


Figure 19-3. Sharing Storage with IARVSERV

Use with Data-in-Virtual (DIV Macro)

There are several restrictions for programs that use data-in-virtual MAP service with data shared using the IARVSERV SHARE service:

- A sharing page must reside in non-swappable storage and have a SHAREDWRITE view mode.
- Only one member of a sharing group can be mapped. Any attempt to map another member of the same sharing group results in a X'08B' abend.
- You cannot use the IARVSERV macro to share data mapped to a hiperspace object.
- You cannot map a sharing page whose sharing group contains a page that is currently a fixed or disabled reference page.

- If the owning address space of a sharing page that was mapped by DIV MAP terminates prior to a DIV UNMAP, the data is lost. Any further reference to the shared data results in a X'028' abend.

There are also restrictions for programs that use the data-in-virtual UNMAP function.

If a sharing page is currently mapped, and the owner of the map issues DIV UNMAP with RETAIN for that page, the value of RETAIN affects all sharing group members as follows:

- For RETAIN=NO, all pages of the target become unpredictable in content.
- For RETAIN=YES, all pages of the target get the data as it last appeared within the sharing page. This can be useful for saving an instance of data, such as a check point. Use of RETAIN=YES can affect performance if it consumes large amounts of central storage by repeated retaining of the storage.

Diagnosing Problems with Shared Data

You can use IPCS reports to see how data is being shared through IARVSERV. The IPCS RSMDATA subcommand with the SHRDATA parameter provides a detailed report on the status of IARVSERV data sharing. The following RSMDATA reports also provide shared data information: ADDRSPACE, EXPFRAME, REALFRAME, RSMREQ, SUMMARY, and VIRTPAGE. See *OS/390 MVS IPCS Commands* for more information about the SHRDATA subcommand.

You may also collect information about data shared through IARVSERV by issuing the DISPLAY command, and by specifying certain optional parameters on the IARR2V macro. See *OS/390 MVS System Commands* and “Converting a Central to Virtual Storage Address (IARR2V Macro)” for more information.

Converting a Central to Virtual Storage Address (IARR2V Macro)

The IARR2V macro provides a simple method to obtain a virtual storage address from a central storage address. This conversion can be useful, for example, when you are working with an I/O or diagnostic program that provides central storage addresses, but you want to use virtual storage addresses.

The details of the syntax and parameters of IARR2V are in *OS/390 MVS Programming: Assembler Services Reference*. In its simplest form, the IARR2V macro requires only the RSA parameter. The RSA parameter specifies the central storage address that you want to convert.

The system returns the virtual storage address in a register or in a storage location you specify through the VSA parameter. Also, you can request the system to return the ASID or STOKEN of the address space or data space associated with the address.

If you require knowledge of whether the central storage address you have is being shared through the IARVSERV macro, you can get that information using the WORKREG, NUMVIEW, and NUMVALID parameters. To use the NUMVIEW and NUMVALID parameters, you must use the WORKREG parameter to specify the work register for the system to use. The NUMVIEW parameter requests the total

number of pages sharing the view of your central storage address. NUMVALID requests the number of pages currently addressable (accessed), which is called the number of valid views. With NUMVIEW and NUMVALID, you can check how effectively programs are using shared storage. Pages that are not accessed have not been read or updated by any program.

Chapter 20. Timing and Communication

This chapter describes timing services and communication services. Use timing services to determine whether the basic or extended time-of-day (TOD) clock is synchronized with an External Time Reference hardware facility (ETR[1]), obtain the present date and time, convert date and time information to various formats, 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/E terminals, and to the system log.

Checking for Timer Synchronization

Several processors can share work in a data processing complex. Each of these processors has access to a TOD clock. Thus, when work is shared among different processors, multiple TOD clocks can be involved. However, these clocks might not be synchronized with one another. The External Time Reference (ETR) is a single external time source that can synchronize the TOD clocks of all processors in a complex.

For programs that are dependent upon synchronized TOD clocks in a multi-system environment, it is important that the clocks are in ETR synchronization. Use the STCKSYNC macro to obtain the TOD clock contents and determine if the clock is synchronized with an ETR. STCKSYNC also provides an optional parameter, ETRID, that returns the ID of the ETR source with which the TOD clock is currently synchronized.

Note: **IBM recommends** the use of the STCKSYNC macro instead of the STCK instruction for all multi-system programs that are dependent upon synchronized clocks.

Obtaining Time of Day and Date

When an ETR is used, the time of day and date are set automatically at system initialization. In other configurations, the operator is responsible for initially supplying the correct time of day and date in terms of a 24-hour clock.

You can use the TIME macro to obtain the time of day and date for programs that require this information. If you specify ZONE=GMT with TIME, the returned time of day and date will be for Greenwich Mean Time. If you specify ZONE=LT or omit the ZONE parameter, the TIME macro returns the local time of day and date. However, if you specify STCK or STCKE , the ZONE parameter has no meaning. When you specify LINKAGE=SYSTEM with the TIME macro, you can select the format for the returned date by using the DATETYPE parameter.

All references to time of day use the time-of-day (TOD) clock, either the basic format (unsigned 64-bit binary number) or the extended format (unsigned 128-bit binary number). The TOD clock runs continuously while the power is on, and the clock is not affected by system-stop conditions. Normally, the clock is reset only when an interruption of processor power has caused the clock to stop, and

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

© Copyright IBM Corp. 1988, 1999

restoration of power has restarted the clock at a later time. When an ETR is used, the clock reset happens automatically; in other configurations, the operator resets the clock. (For more information about the TOD clock, see *Principles of Operation*.)

Converting Between Time of Day and Date and TOD Clock Formats

You can use the STCKCONV macro to convert a TOD clock value to time of day and date, specifying the format in which the information will be returned. This conversion is useful, for example, for producing a report that requires the time and date to be printed in a certain format.

You can use the CONVTOD macro to convert a time of day and date value to TOD or ETOD clock format. The macro accepts a time of day and date value in any of the formats returned by the STCKCONV and TIME macros, and converts that value to either TOD clock format.

It is recommended that you begin to convert your applications to using the ETOD format. The extended time-of-day format was required both to address the time wrapping problem that would occur in the year 2042 and also to provide inproved resolution necessary for the faster processors as they become available.

Note that if you request ETOD information and your processor is not configured with the 128-bit extended time-of-day clock, timer services will return the contents of the 64-bit TOD and will simulate the remaining 64 bits of the ETOD. Conversely, if you request TOD information and your processor is configured with the extended time-of-day clock, timer services will return only that portion of the 128-bit ETOD that corresponds to the 64-bit TOD.

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.

When you cancel a timer request that specified a timer exit:

1. Specify the TU or MIC parameters to determine whether the cancel operation was successful. If the STIMERM or TTIMER macro returns a value of zero to the storage area designated by TU or MIC, then the asynchronous timer completion exit routine has run or will run because its interval expired before the cancel operation completed.
2. It is your responsibility to set up a program to determine whether the timer exit has run; you can have the exit set an indicator to tell you that it has run.

If the STIMERM or TTIMER macro returns a non-zero value to the storage area designated by TU or MIC, then the time interval was cancelled and the asynchronous exit will not run.

Figure 20-1 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 20-1. 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.

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 has used since being created (attached).

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 20-2, 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.

Figure 20-2. Characters Printed or Displayed on an MCS Console

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

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 *OS/390 MVS Programming: Assembler Services Reference* for an explanation of the parameters in the single-line and multiple-line forms of the WTO macro.

Routing the Message

You can route a WTO or WTOR message to a console by specifying one or more of the following keywords:

- ROUTCDE to route messages by routing code
- CONSID to route messages by console ID
- CONSNAME to route messages by console name
- MCSFLAG to route messages by message type.

The ROUTCDE parameter allows you to specify the routing code or codes for a WTO or 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. The installation must define which routing codes are being received by each console.

You can also use either the CONSID or CONSNAME parameter to route messages. These mutually exclusive parameters let you specify a 4-byte field or register that contains the ID or the pointer to a name of the console that is to

receive the message. When you issue a WTO or WTOR macro that uses either the CONSID or CONSNAME parameters with the ROUTCDE parameter, the message or messages will go to all of the consoles specified by both parameters.

The MCSFLAG parameter specifies various attributes of the message, such as whether the message is:

- For a particular console
- For all active consoles
- A command response
- For the hard-copy log.

Specifying MCSFLAG=BUSYEXIT will terminate WTO processing if either of the following conditions occur:

- No console buffers for either MCS or JES3 are available
- There is a JES3 staging area shortage.

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.

Note: For the convenience of the operator, you can associate messages with individual keynames. A keyname consists of 1 to 8 alphanumeric characters, and it appears with the message on the console. The keyname can be used as an operand in the DISPLAY R console command, which operators can issue at the console. Use the KEY parameter on the WTO or WTOR macro for this purpose.

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. 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 20-3 shows the indicator that is used for each descriptor code.

Figure 20-3. Descriptor Code Indicators	
Descriptor Code	**Non-Authorized Problem Program**
1	@
2	@
3-10	blank+
11	@
12-13	blank+

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 *OS/390 MVS Planning: Operations*.

If a 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 *OS/390 MVS 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 20-4.

```
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=(7,8,9),MF=L
```

Figure 20-4. Writing to the Operator

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 system adds a message identifier before the message to associate the reply with the message. The system 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 system 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 system 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 20-5. 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 20-5. 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 system 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 system moves the reply into the issuer's reply area and posts the event control block. Each console that received the original WTOR also receives the accepted reply unless it is a security message. A security message is a WTO or WTOR message with routing code 9. 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

To write a multiple-line message to one or more operator consoles, issue WTO or WTOR with all lines of text.

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.

You can direct a WTO message to a specific console by specifying the console ID or name when issuing the message. For example, you can use the CONSID or CONSNAME parameter on the WTO macro to direct the WTO message to consoles defined by those parameters. If the console is not active anywhere within the sysplex, the system writes the message to the system log unless it is an important information message, an action message or WTOR message. An important information message is a WTO or WTOR message with descriptor codes 1, 2, 3, 11, or 12. 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/E 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 send a message to the system log by coding the information that you wish to log in the “text” parameter of the WTL macro.

The WTO macro with the MCSFLAG=HRDCPY parameter also writes messages to the system log. Because WTO allows you to supply more information on the macro invocation than WTL, **IBM recommends** that you use WTO instead of WTL.

The system 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.

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. If JES3 DLOG is being used, 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 *OS/390 JES3 Commands* for information about the DLOG format of the log entry when using JES3.

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. To delete a message, identify the message by using the MSG, MSGLIST, or TOKEN parameters on the DOM macro, and issue DOM.

When you issued WTO or WTOR to write the message, the system returned a message ID in general purpose register 1. Use this ID as input on the MSG or MSGLIST parameters on the DOM macro. MSGLIST (message list) associates

several message IDs with the delete request. The number of message IDs in the message list is defined by the COUNT parameter or it is defined by a 1 in the high-order bit position of the last message ID in the list. The COUNT parameter cannot exceed 60. If you specified the TOKEN parameter on WTO to generate your own message ID, use the same value on the TOKEN parameter on DOM to delete that message.

Retrieving Console Information (CONVCON Macro)

Programs 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 *OS/390 MVS Data Areas, Vol 1 (ABEP-DALT)* for more information on the CONV parameter list, which is mapped by IEZVG200.

The following topics describe possible uses for the CONVCON macro, and tell you how to fill in the parameter list for each use. The parameter list values are discussed in *OS/390 MVS Programming: Assembler Services Reference*.

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)

   If the console name in CONVFLD is less than 10 characters, pad the name with blanks.

   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 is defined 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 active console is one that is defined and running. 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, CONVSYSN contains the name of the system to which the console is attached, if the console you specified is active. If the console is not active, CONVSYSN contains blanks. Register 15 contains a return code. If you receive the following return codes, check the reason code in CONVRSN for an explanation.

- Return code 0 indicates that the console name or ID is valid and the console is active.
- Return code 4 indicates that the name or ID is valid, but the console is not active.
- Return code 8 indicates that the console name is incorrect.
- Return code 0C indicates that the console ID is incorrect.

See *OS/390 MVS Programming: Assembler Services Reference* for an explanation of all return codes.

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.

Chapter 21. Translating Messages

The MVS message service (MMS) provides a method of translating message text, and provides a convenient method of storing message text.

- MMS enables you to translate U.S. English messages into other languages. These messages can be IBM-supplied messages or application messages. An application can format message text for any language, including English, by issuing the TRANMSG macro.
- MMS enables you to store message text in MMS message files rather than in the application code. By using MMS to store message text, you eliminate the need to include the message text as part of the application code. Any program that needs to issue a particular message can get it from one place: a **run-time message file**. A run-time message file contains messages in a format that MMS can use. You can also update your messages in the **install message files** rather than in the source code. An install message file is a partitioned data set (PDS) that contains **message skeletons**. A message skeleton contains message text and substitution data.

Applications running on TSO/E can have their messages translated automatically if the primary language associated with the TSO/E session is the same language as the language of the run-time message file. A primary language is one that is defined in your TSO/E profile. Therefore, even if you are issuing messages by using the WTO macro, you can present a message in the primary language associated with the TSO/E session. If you are routing system messages to a TSO/E extended MCS console, and MMS is active, users of extended MCS consoles on TSO/E can select available languages for message translation and the system will display translated messages on the user's screen.

Applications based on products not already using MMS must translate their own messages by invoking the TRANMSG macro.

MMS 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.

Preparing IBM-Supplied Messages for Translation: To prepare IBM messages for translation, perform the following tasks:

1. Ensure that the appropriate IBM-supplied system install message files have been installed on your system.

   For MVS messages (MVS, JES2, TSO/E), IBM provides an install message file for U.S. English messages. IBM will also supply Japanese versions of those messages, if requested. When you install MVS, these messages are automatically put into install message files. The U.S. English file is called SYS1.MSGENU.

2. Create a system run-time message file for each language by running the system's install message files through the message compiler. See “Compiling Message Files” on page 21-9 for details on using the compiler.

© Copyright IBM Corp. 1988, 1999

Preparing Application Messages for Translation: To prepare an application's messages for translation, perform the following tasks.

1. Create a PDS for the English version, and a PDS for the translated version of the application's messages. To make it easy to locate and update messages, group messages for each program, component, or other category into separate PDS members. These data sets are the application's install message files. The logical record length of the data set should be variable length of 259, and the block size 23476. **IBM recommends** that you put IBM messages first in a PDS concatenation. If you are not translating IBM messages, you can still use the same recommended logical record length and block size.
2. Validate the application's install message files by running each PDS through the message compiler. See “Compiling Message Files” on page 21-9 for details on using the compiler. The MMS message compiler replaces the entire run-time message file, so create a test run-time message file for each language, using names different from those containing IBM-supplied messages. Creating a test run-time message file enables you to verify the new messages without disturbing the existing system run-time message files and current message translation.
3. After a clean compile, add your PDS members into the system's install message files as new members.
4. Update the system run-time message files by running the system's install message files through the message compiler. See “Updating the System Run-Time Message Files” on page 21-13 for details on updating the system run-time message files.

Figure 21-1 on page 21-3 illustrates the process of preparing messages for translation.

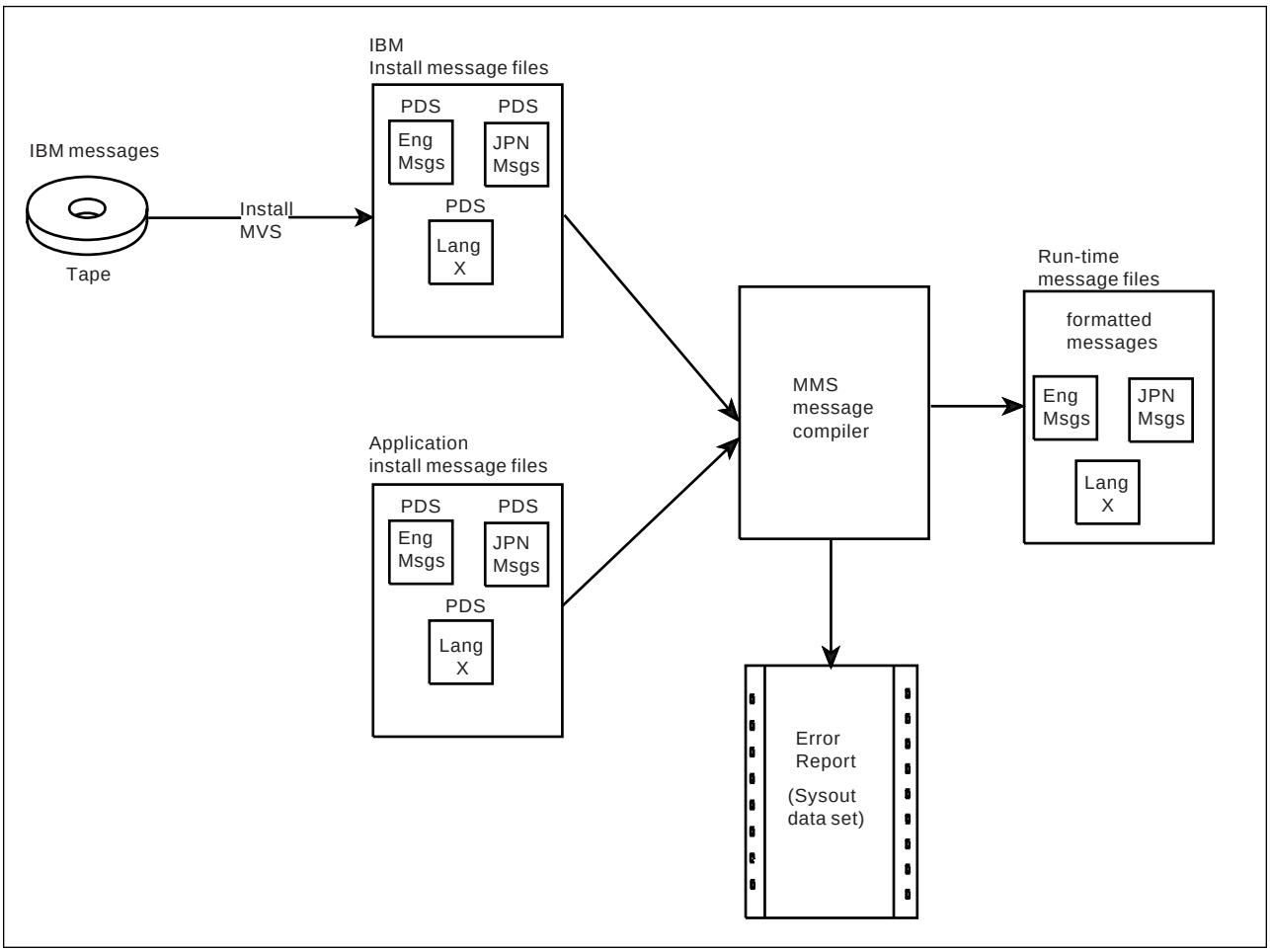


Figure 21-1. Preparing Messages for Translation

Translating Application Messages Using the MVS Message Service: To use MMS in an application, modify the application to exploit the translation service that MMS provides:

- Use the QRYLANG macro to determine which languages are currently available at your installation. For more information on QRYLANG, see “Determining which Languages are Available (QRYLANG Macro)” on page 21-14.
- Use the TRANMSG macro to obtain from MMS the translated version or the complete U.S. English message text of an application's message or messages. For more information on TRANMSG, see “Retrieving Translated Messages (TRANMSG Macro)” on page 21-14.

The installation can translate messages into more than one language. See “Support for Additional Languages” on page 21-18.

Allocating Data Sets for an Application

For an application whose messages will be translated, you must allocate a 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 *OS/390 MVS JCL User's Guide* and *OS/390 MVS 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 21-2. If you are translating MVS messages, you can use the contents of the version record fields for informational purposes. If you are creating messages for an application program, you need only to supply input to columns 1 through 6. Columns 7 through 38 can be blanks.

Figure 21-2. 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	Character field containing a Y or N, indicating whether this language contains a double-byte character set (DBCS).
7-14	Field maintenance identifier (FMID) applicable to the messages within the member, padded on the right with blanks.
15-22	Product identifier applicable to the messages within the member, padded on the right with blanks.
23-38	Service level applicable to the member, padded on the right with blanks.

The following is an example of a version record.

```
.VENUNJBB44N1 5695-047005
```

Figure 21-3 explains the previous example of the version record.

Figure 21-3. Version Record Example

Columns	Example	Description
1 & 2	.V	Version record
3-5	ENU	Three-character language code
6	N	DBCS indicator
7-14	JBB44N1b	FMID
15-22	5695-047	Product identifier
23-38	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.

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 *OS/390 MVS Initialization and Tuning Reference* for more information on these substitution tokens.

Message Skeleton Format

Each message skeleton must follow the column format shown in Figure 21-4.

Figure 21-4 (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 *OS/390 MVS System Messages, Vol 1 (ABA-ASA)* 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. The following is an example of a multi-line message skeleton:
	
	`MSGID01 01 THIS IS LINE ONE OF THIS MULTI-LINE MESSAGE`
	`MSGID01 02 THIS IS LINE TWO OF THIS MULTI-LINE MESSAGE`
	`MSGID01 03 THIS IS LINE THREE OF THIS MULTI-LINE MESSAGE`

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 16 must contain an EBCDIC blank.

Figure 21-4 (Page 2 of 2). Message Skeleton Fields

Columns	Contents and Description

17 & 18	Translated line number (*mm*).
	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`
	You can also use translated line numbers for English message skeletons if your input to the TRANMSG macro is an MPB (message parameter block). In this case, TRANMSG will return all message lines in English for a given message ID.
	If a line of a U.S. English message translates to only one line, leave the translated line number blank.

19	Not part of the intended programming interface.
20 +	Message text. See “Message Text in a Skeleton” on page 21-6

The following are examples of message skeletons.

```
msgid     llfffbmm 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 &X-1 DENIED:
HASP999I  02      USER INFORMED AT &2;=DATE02. ON TERMINAL &X-3
HASP999I  03         LEADING BLANKS ARE OK

IEFA003F    001   USER &USERID VIOLATED ACCESS RIGHTS TO
                        DATASET &X-2 AT &3;=TIME.
IEFA003F    002   &1;=TIME.: USER &X-2 VIOLATED ACCESS RIGHTS TO DATASET &X-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 255 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. Blanks are valid characters, and are acceptable as any part of the message text. Message text can contain substitution tokens.

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;
```

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 *OS/390 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 run-time message files, 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 formatted run-time message files. Before invoking the compiler, you must allocate storage for these run-time files, but you cannot use them 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 Validation Run-Time Message Files

The data set you create for the run-time message files 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 files, you need to specify the DEFINE CLUSTER function of access method services (IDCAMS) with the LINEAR parameter. When you code the SHAREOPTIONS parameter for DEFINE CLUSTER, use SHAREOPTIONS (1,3). For a complete explanation of SHAREOPTIONS, see *DFSMS/MVS Using Data Sets*.

The following is a sample job that invokes 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 21-5. Sample job to invoke IDCAMS to obtain a data set for the run-time message files

Compiling Message Files

The message compiler creates run-time message files 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 or a concatenation of PDSs 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 21-12 for a description of compiler return codes.

Invoking the Message Compiler

The message compiler is an executable program. You can use JCL, 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 21-6. Using JCL to Invoke the Compiler with a single PDS as input

```
//COMPILE  EXEC  PGM=CNLCCPLR,
//               PARM=(lang,dbcs)
//SYSUT1   DD    DSN=msg_pds1,DISP=SHR
//         DD    DSN=msg_pds2,DISP=SHR
           :
           :
//         DD    DSN=msg_pdsn,DISP=SHR
//SYSUT2   DD    DSN=msg_div_obj,DISP=(OLD,KEEP,KEEP)
//SYSPRINT DD    SYSOUT=*
```

Figure 21-7. Using JCL to Invoke the Compiler with a concatenation of partitioned Data Sets as input

```
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 21-8. Using a TSO/E CLIST to Invoke the Compiler with a single PDS input

```
PROC 0
FREE DD(SYSUT1,SYSUT2,SYSPRINT)             /* FREE DD'S           */
ALLOC DD(SYSUT1) DSN('msg_pds1' +            /* ALLOC INPUT FILE   */
ALLOC DD(SYSUT1) DSN 'msg_pds1' +            /* ALLOC INPUT FILE   */
                       :
                       :
ALLOC DD(SYSUT1) DSN 'msg_pdsn') 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 21-9. Using a TSO/E CLIST to Invoke the Compiler with a concatenation of partitioned Data Set as input

```
/* 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(SYSUT1,SYSUT2,SYSPRINT)"

return(compiler_rc)
```

Figure 21-10. Using a REXX exec to Invoke the Compiler with a single PDS as input

```
/* MESSAGE COMPILER INVOCATION EXEC */

MSGCMPLR:

"FREE DD(SYSUT1,SYSUT2,SYSPRINT)"

"ALLOC DD(SYSUT1) DSN('msg_pds1',",
                      "'msg_pds2',",
                            :
                            :
                      "'msg_pdsn') 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 21-11. Using a REXX exec to Invoke the Compiler with a concatenation of partitioned Data Sets as input

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_pds1
is the name of the install message file containing the the first application's message skeletons for a specific language. *msg_pds1* must be a partitioned data set.

msg_pds2
is the name of the install message file containing the the second application's message skeletons for a specific language. *msg_pds2* must be a partitioned data set.

msg_pdsn
is the name of the install message file containing the the last application's message skeletons, in the message skeleton PDS concatenation, for a specific language. *msg_pdsn* must be a partitioned data set.

Note: When you specify a concatenation of partitioned data set as input to the MVS message service (MMS) compiler, all members within the partitioned data set will be processed. The MMS compiler will process all members within the concatenation of partitioned data sets without regard to uniqueness of member names. If two partitioned data sets specified in the concatenation have members with the same name, both members will be processed by the MMS compiler.

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 run-time message files by compiling the install message file, determine the amount of storage the run-time message files used. This calculation is necessary when compiling these messages in the system's run-time message files. 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 in register 15. The return codes are as follows:

Code	Meaning
0	Successful completion.
4	Processing complete. Run-time message files are complete but the compiler generated warnings.

Code	Meaning
8	Processing complete. The run-time message files are usable but incomplete.
12	Processing ended prematurely. The run-time message files are unusable.

You should correct all errors and recompile if you receive any return code other than 0.

Updating the System Run-Time Message Files

After validating your application install message files, update the system run-time message files. The TRANMSG macro can retrieve messages from the run-time message files. You need to do the following:

- Add the application's install message files to the system's install message files, or add a DD statement to the JCL used to compile the system's install message files.
- Allocate a data set for the new system run-time message files. Assign unique names to the run-time message files, ensuring that the names are different from those your installation is currently using. Use the storage requirements you received from running the IDCAMS report.
- Compile the system's install message files into new system run-time message files using the message compiler for each install message file. See “Compiling Message Files” on page 21-9.
- Identify your new run-time message files to the system by creating a new MMSLSTxx parmlib member. See *OS/390 MVS Initialization and Tuning Reference* for information on creating a parmlib member.
- Activate your new parmlib member and run-time message files by issuing the SET MMS=xx command. See *OS/390 MVS System Commands* for information on the SET MMS=xx command.

You are using the same invocations to update the system run-time message files as you do to verify message skeletons. The difference is that the resulting system run-time message files are 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 from the system run-time message files. You need to do the following:

- Determine the language in which you want the application's messages translated, and use the QRYLANG macro to check its availability.
- Retrieve translated messages using the TRANMSG macro.

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 Macro)

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 CNLMLQB.
- Determine the total number of languages by looking in the MCAALCNT (active language count) field of the MCA, mapped by CNLMMCA. Your program must be in 31-bit addressing mode to reference the MCA.
- 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 CNLMLQB.
- Add the length of the LQB substitution data area to the length of the header.

Retrieving Translated Messages (TRANMSG Macro)

TRANMSG takes the application messages that you provide, and retrieves the corresponding translated messages from the system run-time message files. TRANMSG returns the translated message in a message text block (MTB).

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 Macro).”

In your application, call TRANMSG as close to the point of message presentation as possible to avoid presenting a translated version of the message to MVS functions (for example, installation exits, automation CLISTs, MCS consoles) that expect English text.

A message input/output block (MIO), mapped by CNLMMIO, serves as both input to and output from 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 by using the information you supply through the macro parameters.

Build the MIO yourself if you are translating multi-line messages that have continuation lines. You will need to set the MIOCONT flag in the MIO.

If you build the MIO yourself, the MIO must contain the following when you issue TRANMSG:

- 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.

See *OS/390 MVS Data Areas, Vol 3 (IVT-RCWK)* for a mapping of the MIO.

The application's input 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 contains the address of the translated message in the output buffer. The translated message is in the form of an MTB.

Translating a multi-line message is a little different from translating a single-line message. You must take one of the following steps in preparing the multi-line message for translation:

- Add the message identifier to the beginning of the message text for each line subsequent to the first. You can invoke TRANMSG once for all message lines, or once for each message line. 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 following example). The MIOCONT flag informs 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. *OS/390 MVS Programming: Assembler Services Reference* provides a coding example that translates a multi-line message.

  The following is an example of a multi-line message that contains continuation lines, that is, only the first line contains a message identifier in the skeleton. You must include all lines in the TRANMSG invocation.

  ```
  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 21-12 on page 21-16 shows how an application program uses the TRANMSG macro.

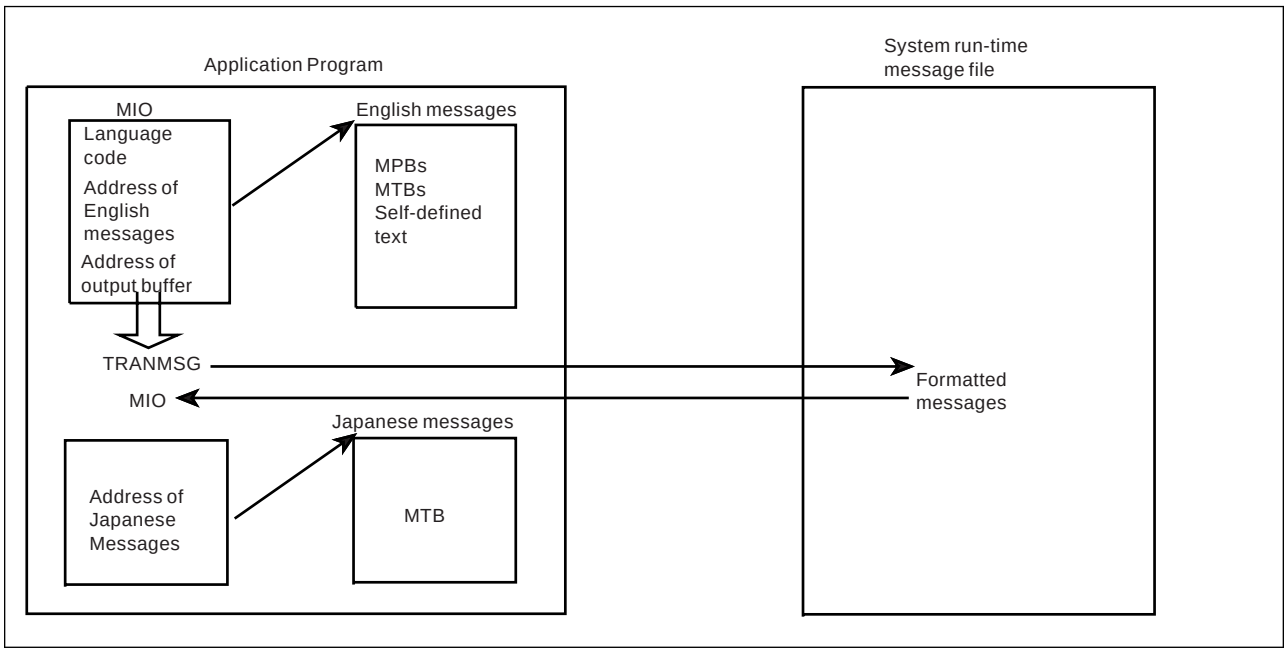


Figure 21-12. Using the TRANMSG Macro

Example of Displaying Messages

The following table shows two ways a general application program (a program not using WTO to issue messages) can display messages.

Displaying a message without using MMS	Displaying a message using MMS
• Build a buffer for an English message containing a message ID, fixed text, and substitution data. • Display the English message (for example, by using TSO/E PUTLINE). The application supports only English.	• Build a buffer for an English message. You must match the message format of the English message buffer to the format of the English message skeleton. • Issue the TRANMSG macro using the English message buffer as input, and getting a translated message buffer as output. • Present the translated message, ensuring that your output device supports the chosen language. By "presenting" a message, you might choose to write the message to a data set, or write it to a screen. **OR** • Build an MPB. See "Using Message Parameter Blocks for New Messages (BLDMPB and UPDTMPB Macros)" on page 21-17 for information on creating and MPB. • Issue the TRANMSG macro using an MPB as input. TRANMSG will format the substitution data contained in the MPB. The output is a Japanese message buffer (if language code is JPN). You should not use WTO to issue the translated message.

Using Message Parameter Blocks for New Messages (BLDMPB and UPDTMPB Macros)

You can create message parameter blocks (MPBs) instead of storing messages in your application code. MPBs contain a message identifier, and, if needed, a format number, line number, and any substitution data. The actual message text resides only in the message skeletons in the run-time message file. Using MPBs 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.

If you use message text blocks (MTBs) or self-defined text as input to the TRANMSG macro, and your message text requires a change, you will have to change it in both the message skeleton and the MTB or self-defined text. Modifying just the existing run-time message files to adapt changed message text can result in unpredictable errors when using the TRANSMSG service.

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 MPB. 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 21-13 that matches the language into which you plan to translate messages. Though you may supply your own language code, IBM recommends that you use these codes. You will need that language code for the version record. Figure 21-13 is not an all-inclusive list of languages.
- 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 21-4, “Validating Message Skeletons” on page 21-7, and “Updating the System Run-Time Message Files” on page 21-13.
- 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 *OS/390 MVS Initialization and Tuning Reference* for more information on setting up config members, and parmlib members.

Figure 21-13 (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

Figure 21-13 (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
ENU	US English	United States
ESP	Spanish	Spain
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 (BLDMPB, QRYLANG, TRANMSG, UPDTMPB) in the *OS/390 MVS Programming: Assembler Services 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
         CNLMMPB
         CNLMMCA
         CNLMMIO
VARS     DSECT
VARSINBF DS    F
VARSAREA DS    CL24
VARSLEN  EQU   *-VARS
         END TRANSMPB
```

Chapter 22. Data Compression and Expansion Services

MVS/ESA supports compression using two different algorithms. The first algorithm exploits the occurrence of repeated characters in a data stream. Encoded data contains a combination of symbols that represent strings of repeated characters and the original characters that are not repeated. The CSRCESRV service of MVS/ESA uses this algorithm to compress data.

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.

The second algorithm encodes data by replacing strings of characters with shorter fixed-length symbols. A key component of this technique is the symbol table, usually referred to as a dictionary. Encoded data contains symbols that correspond to entries in the dictionary. The CSRCMPSC service uses this algorithm to compress data.

This service requires more storage than the CSRCESRV services but is a good choice if you know what your data looks like and storage is not a concern.

Services Provided by CSRCESRV

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 which is a compression technique that compresses repeating characters, such as blanks. 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 22-3 for further details.

- **Query Service**

  This service queries the system to determine the following:

© Copyright IBM Corp. 1988, 1999

- 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.

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 22-3 for further details.

Figure 22-1 summarizes the services available, and the systems under which these services can run.

Figure 22-1. Summary of Data Compression and Expansion Services

SYSTEM	**MVS/ESA Version of Service**	**MVS/XA Version of Service**
MVS/ESA	Compression Expansion Query	Expansion Query
MVS/XA	Not applicable	Expansion Query

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 *OS/390 MVS Programming: Assembler Services 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” 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 22-2 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 22-2. 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 *OS/390 MVS 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 *OS/390 MVS Programming: Assembler Services 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 *OS/390 MVS Programming: Assembler Services 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.

Services Provided by CSRCMPSC

Compression takes an input string of data and, using a data area called a *dictionary*, produces an output string of compression symbols. Each symbol represents a string of one or more characters from the input.

Expansion takes an input string of compression symbols and, using a *dictionary*, produces an output string of the characters represented by those compression symbols.

Parameters for the CSRCMPSC macro are in an area mapped by DSECT CMPSC of the CSRYCMPS macro and specified by the CBLOCK parameter of the CSRCMPSC macro. This area contains such information as:

- The address, ALET, and length of a source area. The source area contains the data to be compressed for a compression operation, or to be expanded for an expansion operation.
- The address, ALET, and length of a target area. After the macro runs, the target area contains the compressed data for a compression operation, or the expanded data for an expansion operation.
- An indication of whether to perform compression or expansion.
- The address and format of a dictionary to be used to perform the compression or expansion. The dictionary must be in the same address space as the source area.

Compressing and expanding data is described in the following topics:

- “Compression and Expansion Dictionaries” on page 22-5
- “Building the CSRYCMPS Area” on page 22-5

- "Determining if the CSRCMPSC Macro Can Be Issued on a System" on page 22-8

To help you use the compression services, the SYS1.SAMPLIB system library contains the following REXX execs:

- CSRBDICT for building example dictionaries
- CSRCMPEX for measuring the degree of compression that the dictionaries provide

When running on VM, two analogous macros are available:

- CSRBDICV for building example dictionaries
- CSRCMPEV for measuring the degree of compression that the dictionaries provide

The prologs of the execs tell how to use them. For additional information about compression and using the execs, see *Enterprise Systems Architecture/390 Data Compression*.

Compression and Expansion Dictionaries

To accomplish compression and expansion, the macro uses two dictionaries: the compression dictionary and the expansion dictionary. These dictionaries are related logically and physically. When you expand data that has been compressed, you want the result to match the original data. Thus the dictionaries are complementary. When compression is done, the expansion dictionary must immediately follow the compression dictionary, because the compression algorithm examines entries in the expansion dictionary.

Each dictionary consists of 512, 1024, 2048, 4096, or 8192 8-byte entries and begins on a page boundary. When the system determines or uses a compression symbol, the symbol is 9, 10, 11, 12, or 13 bits long, with the length corresponding to the number of entries in the dictionary. Specify the size of the dictionary in the CMPSC_SYMSIZE field of the CSRYCMPS mapping macro:

SYMSIZE	Meaning
1	Symbol size 9 bits, dictionary has 512 entries
2	Symbol size 10 bits, dictionary has 1024 entries
3	Symbol size 11 bits, dictionary has 2048 entries
4	Symbol size 12 bits, dictionary has 4096 entries
5	Symbol size 13 bits, dictionary has 8192 entries

The value of CMPSC_SYMSIZE represents the size of the compression and expansion dictionaries. For example, if CMPSC_SYMSIZE is 512, then the size of the compression dictionary is 512 and the size of the expansion dictionary is 512.

Building the CSRYCMPS Area

The CSRYCMPS area is mapped by the CSRYCMPS mapping macro and is specified in the CBLOCK parameter of the CSRCMPSC macro. The area consists of 7 words that should begin on a word boundary. Unused bits in the first word must be set to 0.

- Set 4-bit field CMPSC_SYMSIZE in byte CMPSC_FLAGS_BYTE2 to a number from 1 to 5 to indicate both the number of entries in the dictionary and the size of a compressed symbol.
- If expanding, turn on bit CMPSC_EXPAND in byte CMPSC_FLAGS_BYTE2. Otherwise, make sure that the bit is off.
- Set field CMPSC_DICTADDR to the address of the necessary dictionary. If compressing, this should be the compression dictionary, which must be **immediately followed** by the expansion dictionary. If expanding, this should be the expansion dictionary. In either case, the dictionary must begin on a page boundary, as the low order 12 bits of the address are assumed to be 0 when determining the address of the dictionary.

  If running in AR mode, set field CMPSC_SOURCEALET to the ALET of the necessary dictionary. Note that the input area is also accessed using this ALET. If not in AR mode, make sure that the field contains 0.

  Using the linkage editor can help you get the dictionary on the proper boundary. For example, you may have an object deck for each compression dictionary (CD), and expansion dictionary (ED), and a DD statement for file OBJS, which represents the library containing the object decks. The following instructions linkage editor control statements define a dictionary with the name DICT that, when loaded, will have the compression dictionary followed by the expansion dictionary, and will begin on a page boundary.

  ```
  ORDER CD(P),ED
  INCLUDE OBJS(CD)
  INCLUDE OBJS(ED)
  NAME DICT(R)
  ```

- In most cases, make sure that 3-bit field CMPSC_BITNUM in byte CMPSC_DICTADDR_BYTE3 is zero. This field has the following meaning:
  - If compressing, place the first compression symbol at this bit in the leftmost byte of the target operand. Normally this field should be set to 0 for the start of compression.
  - If expanding, expand beginning with the compression symbol that begins with this bit in the leftmost byte of the source operand. Normally this field should be set to the value used for the start of compression.
- Set word CMPSC_TARGETADDR to the address of the output area. For compression, the output area contains the compressed data; for expansion, it contains the expanded data.

  If running in AR mode, set field CMPSC_TARGETALET to the ALET of the output area. If not in AR mode, make sure that the field contains 0.
- Set word CMPSC_TARGETLEN to the length of the output area.
- Set word CMPSC_SOURCEADDR to the address of the input area. For compression, the input area contains the data to be compressed; for expansion, it contains the compressed data.

  If running in AR mode, set field CMPSC_SOURCEALET to the ALET of the input area. Note that the dictionary will also be accessed using this ALET. If not in AR mode, make sure that the field contains 0.
- Set word CMPSC_SOURCELEN to the length of the input area. For expansion, the length should be the difference between CMPSC_TARGETLEN at the completion of compression and CMPSC_TARGETLEN at the start of

compression, increased by 1 if field CMPSC_BITNUM was nonzero upon completion of compression.

- Set word CMPSC_WORKAREAADDR to the address of a 192-byte work area for use by the CSRCMPSC macro. The work area should begin on a doubleword boundary. This area does not need to be provided and the field does not have to be set if your code has verified that the hardware CMPSC instruction is present. The program can do the verification by checking that bit CVTCMPSH in mapping macro CVT is on.

When the CSRCMPSC service returns, it has updated the input CSRYCMPS area as follows:

- CMPSC_FLAGS is unchanged.
- CMPSC_DICTADDR is unchanged, but bits CMPSC_BITNUM in field CMPSC_DICTADDR_BYTE3 are set according to the last-processed compression symbol.
- CMPSC_TARGETADDR is increased by the number of output bytes processed.
- CMPSC_TARGETLEN is decreased by the number of output bytes processed.
- CMPSC_SOURCEADDR is increased by the number of input bytes processed.
- CMPSC_SOURCELEN is decreased by the number of input bytes processed.
- CMPSC_WORKAREA is unchanged.

The target/source address and length fields are updated analogously to the corresponding operands of the MVCL instruction, so that you can tell upon completion of the operation how much data was processed and where you might want to resume if you wanted to continue the operation.

Suppose that you had compressed a large area but wanted to expand it back into a small area of 80-byte records. You might do the expansion as follows:

```
         LA   2,MYCBLOCK
         USING CMPSC,2
         XC   CMPSC(CMPSC_LEN),CMPSC
         OI   CMPSC_FLAGS_BYTE2,CMPSC_SYMSIZE_1
         OI   CMPSC_FLAGS_BYTE2,CMPSC_EXPAND
         L     3,EDICTADDR             Address of expansion dictionary
         ST    3,CMPSC_DICTADDR        Set dictionary address
         L     3,EXPADDR
         ST    3,CMPSC_SOURCEADDR      Set compression area
         L     3,EXPLEN
         ST    3,CMPSC_SOURCELEN       Set compression length
         LA    3,WORKAREA
         ST    3,CMPSC_WORKAREAADDR    Set work area address
MORE     DS    0H                      Label to continue
*
* Your code to allocate an 80-byte output area would go here
*
         ST   x,CMPSC_TARGETADDR      Save target expansion area
         LA   3,80                    Set its length
         ST   3,CMPSC_TARGETLEN       Set expansion length
         CSRCMPSC CBLOCK=CMPSC        Expand
         C    15,=AL4(CMPSC_RETCODE_TARGET) Not done, target used up
         BE   MORE                    Continue with operation
         DROP 2
 .
 .
         DS   0F        Align parameter on word boundary
MYCBLOCK DS   (CMPSC_LEN)CL1    CBLOCK Parameter
EXPADDR  DS   A         Input expansion area
EXPLEN   DS   F         Length of expansion area
EDICTADDR DS  A         Address of expansion dictionary
         DS   0D        Doubleword align work area
WORKAREA DS   CL192     Work area
         CSRYCMPS ,     Get mapping and equates
```

Note that this code loops while the operation is not complete, allocating a new 80-byte output record. It does not have to update the CMPSC_BITNUM, CMPSC_SOURCEADDR, or CMPSC_SOURCELEN fields, because the service sets them up for continuation of the original operation.

If running in AR mode, the example would also have set the CMPSC_TARGETALET and CMPSC_SOURCEALET fields. The XC instruction zeroed those fields as needed when running in primary ASC mode.

Determining if the CSRCMPSC Macro Can Be Issued on a System

Do the following to tell if the system contains the software or hardware to run a CSRCMPSC macro:

1. **Determine if CSRCMPSC is available**, by running the following:

```
         L    15,16                  Get CVT address
         USING CVT,15                Set up addressability to the CVT
         TM CVTFLAG2,CVTCMPSC        Is CSRCMPSC available?
         BZ NO_CSRCMPSC              Branch if not available
* Compression feature is available
  .
  .
NO_CSRCMPSC DS 0H
```

2. **Determine if the CMPSC hardware instruction is available**, by running the following:

```
         L    15,16                  Get CVT address
         USING CVT,15                Set up addressability to the CVT
         TM CVTFLAG2,CVTCMPSH        Is CMPSC hardware available?
         BZ NO_CMPSC_HARDWARE        Branch if not available
* CMPSC hardware is available
  .
  .
NO_CMPSC_HARDWARE DS 0H
```

The remaining three topics in this chapter describe compression and expansion processing and the dictionary entries in much greater detail. If you plan to use the CSRBDICT exec to build your dictionary, you do not need to read these topics. If you plan to build your own dictionary, you will want to read:

- "Compression Processing"
- "Expansion Processing" on page 22-10
- "Dictionary Entries" on page 22-10

Compression Processing

The compression dictionary consists of a specified number of 8-byte entries. The first 256 dictionary entries correspond to the 256 possible values of a byte and are referred to as *alphabet entries*. The remaining entries are arranged in a downward tree, with the alphabet entries being the topmost entries in the tree. That is, an alphabet entry may be a *parent entry* and contain the index of the first of one or more contiguous *child entries*. A child entry may, in turn, be a parent and point to its own children. Each entry may be identified by its *index*, meaning the positional number of the entry in the dictionary; the first entry has an index of 0.

An alphabet entry represents one character. A nonalphabet entry represents all of the characters represented by its ancestors and also one or more additional characters called *extension characters*. For compression, the system uses the first character of an input string as an index to locate the corresponding alphabet entry. Then the system compares the next character or characters of the string against the extension character or characters represented by each child of the alphabet entry until a match is found. The system repeats this process using the children of the last matched entry, until the last possible match is found, which might be a match on only the alphabet entry. The system uses the index of the last matched entry as the compression symbol.

The first extension character represented by a child entry exists as either a child character in the parent or as a sibling character. A parent can contain up to four or five child characters. If the parent has more children than the number of child characters that can be in the parent, a dictionary entry named a *sibling descriptor* follows the entry for the last child character in the parent. The sibling descriptor can

contain up to six additional child characters, and a dictionary entry named a sibling descriptor extension can contain eight more child characters for a total of fourteen. These characters are called *sibling characters*. The corresponding additional child entries follow the sibling descriptor. If necessary, another sibling descriptor follows the additional child entries, and so forth. The dictionary entries that are not sibling descriptors or sibling descriptor extensions are called character entries.

If a nonalphabet character entry represents more than one extension character, the extension characters after the first are in the entry; they are called additional extension characters. The first extension character exists as a child character in the parent or as a sibling character in a sibling descriptor or sibling descriptor extension. The nonalphabet character entries represent either:

- If the entry has no children or one child, from one to five extension characters.
- If the entry has more than one child, one or two extension characters. If the entry represents one extension character, it can contain five child characters. If it represents two extension characters, it can contain four child characters.

Expansion Processing

The dictionary used for expansion also consists of a specified number of 8-byte entries. The two types of entries used for expansion are:

- Unpreceded entries
- Preceded entries

The compression symbol, which is an index into the dictionary, locates that index's dictionary entry. The symbol represents a character string of up to 260 characters. If the entry is an unpreceded entry, the expansion process places at offset 0 from the current processing point the characters designated by that entry. Note that the first 256 correspond to the 256 possible values of a byte and are assumed to designate only the single character with that byte value.

If the entry is a preceded entry, the expansion process places the designated characters at the specified offset from the current processing point. It then uses the information in that entry to locate the preceding entry, which may be either an unpreceded or a preceded entry, and continues as described previously.

The sibling descriptor extension entries described earlier are also physically located within the expansion dictionary.

Dictionary Entries

The following notation is used in the diagrams of dictionary entries:

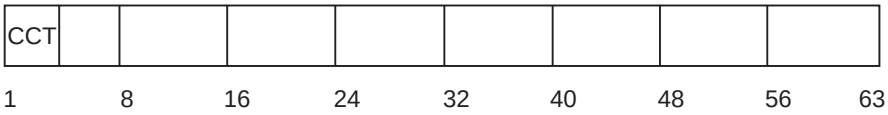
{cc}	Character may be present
...	The preceding field may be repeated

Compression Dictionary Entries

Compression entries are mapped by DSECTs in macro CSRYCMPD.

The first four entries that follow give the possible values for bits 0-2, which are designated CCT.

Character Entry Generic Form (DSECT CMPSCDICT_CE)



CCT A 3-bit field (CMPSCDICT_CE_CHILDCT) specifying the number of children. The total number of children plus additional extension characters is limited to 5. If this field plus the number of additional characters is 6, it indicates that, in addition to the maximum number of children for this entry, there is a sibling descriptor entry that describes additional children. The sibling descriptor entry is located at dictionary entry CMPSCDICT_CE_FIRSTCHILDINDEX plus the value of CMPSCDICT_CE_CHILDCT. The value of CMPSCDICT_CE_CHILDCT plus the number of additional extension characters must not exceed 6.

Character Entry CCT=0 (DSECT CMPSCDICT_CE)

ACT A 3-bit field (CMPSCDICT_CE_AECCT) indicating the number of additional extension characters in the entry. Its value must not exceed 4. This field must be 0 in an alphabet entry.

EC An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

Character Entry CCT=1 (DSECT CMPSCDICT_CE)

XXXXX

A 5-bit field (CMPSCDICT_CE_EXCHILD) with the first bit indicating whether it is necessary to examine the character entry for the child character (looking either for additional extension characters or more children). The other bits are ignored when CCT=1.

ACT A 3-bit field (CMPSCDICT_CE_AECCT) indicating the number of additional extension characters. Its value must not exceed 4. This field must be 0 in an alphabet entry.

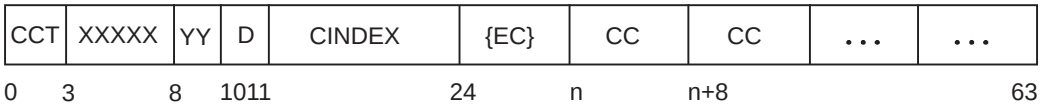
CINDEX

A 13-bit field (CMPSCDICT_CE_FIRSTCHILDINDEX) indicating the index of the first child. The index for child *n* is then CMPSCDICT_CE_FIRSTCHILDINDEX + n-1.

EC An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

CC Child character, at bit n = 24 + (ACT * 8). The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

Character Entry CCT>1 (DSECT CMPSCDICT_CE)



CCT A 3-bit field (CMPSCDICT_CE_CHILDCT) specifying the number of children. For this case, because CCT>1, the range for CCT is 2 to 6 if D=0 or 2 to 5 if D=1. If this field plus the value of D is 6, it indicates that, in addition to the maximum number of children for this entry (4 if D=1, 5 if D=0), there is a sibling descriptor entry that describes additional children. The sibling descriptor entry is located at dictionary entry CMPSCDICT_CE_FIRSTCHILDINDEX plus the value of CMPSCDICT_CE_CHILDCT.

XXXXX
A 5-bit field (CMPSCDICT_CE_EXCHILD) with a bit for each child in the entry. The field indicates whether it is necessary to examine the character entry for the child character (looking either for additional extension characters or more children). The bit is ignored if the child does not exist.

YY A 2-bit field (CMPSCDICT_CE_EXSIB) providing examine-child bits for the 13th and 14th siblings designated by the first sibling descriptor for children of this entry. The bit is ignored if the child does not exist. Note that this is a subfield of CMPSCDICT_CE_AECCT. Do not set both this field and field CMPSCDICT_CE_AECCT in a character entry.

D A 1-bit field (CMPSCDICT_CE_ADDEXTCHAR) indicating whether there is an additional extension character. Note that this is a subfield of CMPSCDICT_CE_AECCT. Do not set both this field and field CMPSCDICT_CE_AECCT in a character entry. This bit must be 0 in an alphabet entry.

CINDEX
A 13-bit field (CMPSCDICT_CE_FIRSTCHILDINDEX) indicating the index of the first child. The index for child *n* is CMPSCDICT_CE_FIRSTCHILDINDEX + n-1.

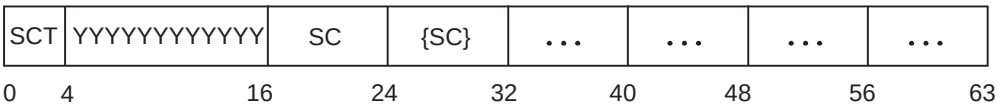
EC An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension character followed by the child characters. There is no additional extension character if D=0.

CC Child character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters. The first child character is at bit n = 24 + (D * 8).

Alphabet Entries (DSECT CMPSCDICT_CE)

The alphabet entries have the same mappings as character entries but without the additional extension characters. The character entries are “Character Entry Generic Form (DSECT CMPSCDICT_CE)” on page 22-11, “Character Entry CCT=0 (DSECT CMPSCDICT_CE)” on page 22-11, “Character Entry CCT=1 (DSECT CMPSCDICT_CE)” on page 22-11, and “Character Entry CCT>1 (DSECT CMPSCDICT_CE)” on page 22-12.

Format 1 Sibling Descriptor (DSECT CMPSCDICT_SD)



SCT A 4-bit field (CMPSCDICT_SD_SIBCT) specifying the number of sibling characters. The number of sibling characters is limited to 14. If this field is 15, it indicates that there are 14 sibling characters associated with this entry and that there is another sibling descriptor entry, which describes additional children. That sibling descriptor entry is located at dictionary entry this-sibling-descriptor-index + 15. If there are 1 to 6 sibling characters, they are contained in this entry, and the dictionary entries for those characters are located at this-sibling-descriptor-index + n, where n is 1 to 6. If there are 7 to 14 sibling characters, the first 6 are as described above, and characters 7 through 14 are located in the **expansion** dictionary entry. (See “Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)” on page 22-14.) The index of the character entry is this-sibling-descriptor-index. The number of sibling characters should not be 0.

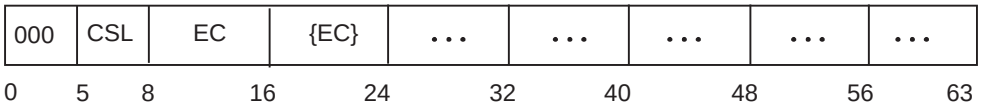
YYYYYYYYYYYY
A 12-bit field (CMPSCDICT_SD_EXSIB), one for each sibling character, indicating whether to examine the character entries for sibling characters 1 through 12. Recall that the examine-sibling indicator for sibling characters 13 and 14 for the first sibling descriptor is in the character entry field CMPSCDICT_CE_EXSIB. If this is not the first sibling descriptor for the child entry, then the character entries for sibling characters 13 and 14 are examined irregardless. The bit is ignored if the sibling does not exist.

SC Sibling character. Sibling characters 8 through 14 are in the expansion dictionary. (See “Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)” on page 22-14.) The 6-character field (CMPSCDICT_SD_CHILDCHAR) is provided to contain the sibling characters. The index of the character entry for sibling character *n* is this-sibling-descriptor-index + n-1.

Expansion Dictionary Entries

Expansion entries are mapped by DSECTs in macro CSRYCMPD.

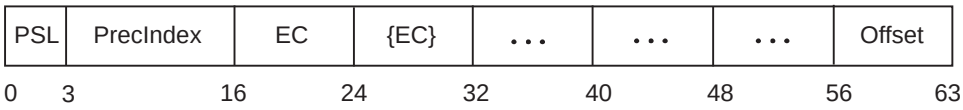
Unpreceded Entry (DSECT CMPSCDICT_UE)



CSL A 3-bit field (CMPSCDICT_UE_COMPSYMLEN) indicating the number of characters contained in CMPSCDICT_UE_CHARS. These characters will be placed at offset 0 in the expanded output. This field should not have a value of 0.

EC Expansion character. The 7-character field (CMPSCDICT_UE_CHARS) is provided to contain the expansion characters.

Preceded Entry (DSECT CMPSCDICT_PE)



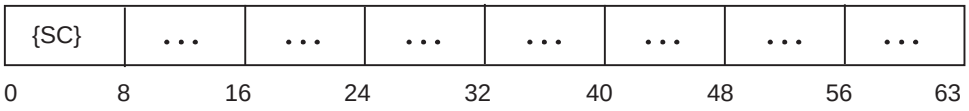
PSL A 3-bit field (CMPSCDICT_PE_PARTSYMLEN) indicating the number of characters contained in CMPSCDICT_PE_CHARS. These characters will be placed at the offset indicated by CMPSCDICT_PE_OFFSET in the expanded output. This field must not be 0, because 0 indicates an unpreceded entry.

PrecIndex
A 13-bit field (CMPSCDICT_UE_PRECENTINDEX) indicating the index of the dictionary entry with which processing is to continue.

EC Expansion character. The 5-character field (CMPSCDICT_PE_CHARS) is provided to contain the expansion characters.

Offset A 1-byte field (CMPSCDICT_PE_OFFSET) indicating the offset in the expanded output for characters in CMPSCDICT_PE_CHARS.

Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)



SC Sibling character. The 8-character field (CMPSCDICT_SDE_CHARS) is provided to contain the sibling characters. The nth sibling character in this entry is actually overall sibling character number 6 + n, because the first 6 characters were contained in the corresponding sibling descriptor entry. The index of the character entry for the nth character is this-sibling-descriptor-index + 6 + n-1.

Dictionary Restrictions

Set up the compression dictionary so that:

- The algorithm does not create a compression symbol that represents a string of more than 260 characters.
- No character entry has more than 260 total children, including all sibling descriptors for that character entry.
- No character entry has a child count greater than 6.
- No character entry has more than 4 additional extension characters when there are 0 or 1 child characters.
- No sibling descriptor indicates 0 sibling characters.

Set up the expansion dictionary so that:

- Expansion of a compression symbol does not use more than 127 dictionary entries.

Other Considerations

If the first child character matches, but its additional extension characters do not match and the next child character is the same as the first, the system continues compression match processing to try to find a compression symbol that contains that child character. If, however, the next child character is not the same, compression processing uses the current compression symbol as the result. You can set up the child characters for an entry to take advantage of this processing.

If a parent entry does not have the examine child bit (CMPSCDICT_CE_EXCHILD) on for a particular child character, then the child character entry should not have any additional extension characters or children. The system will not check the entry itself for additional extension characters or children.

If a parent or sibling descriptor entry does not have the examine sibling bit (CMPSCDICT_CE_EXSIB) on for a particular sibling character, then the character entry for that sibling character should not to have any additional extension characters or children. The system will not check the entry itself for additional extension characters or children.

Compression Dictionary Examples

In the following examples, most fields contain their hexadecimal values. However, for clarity, the examine-child bit fields are displayed with their bit values.

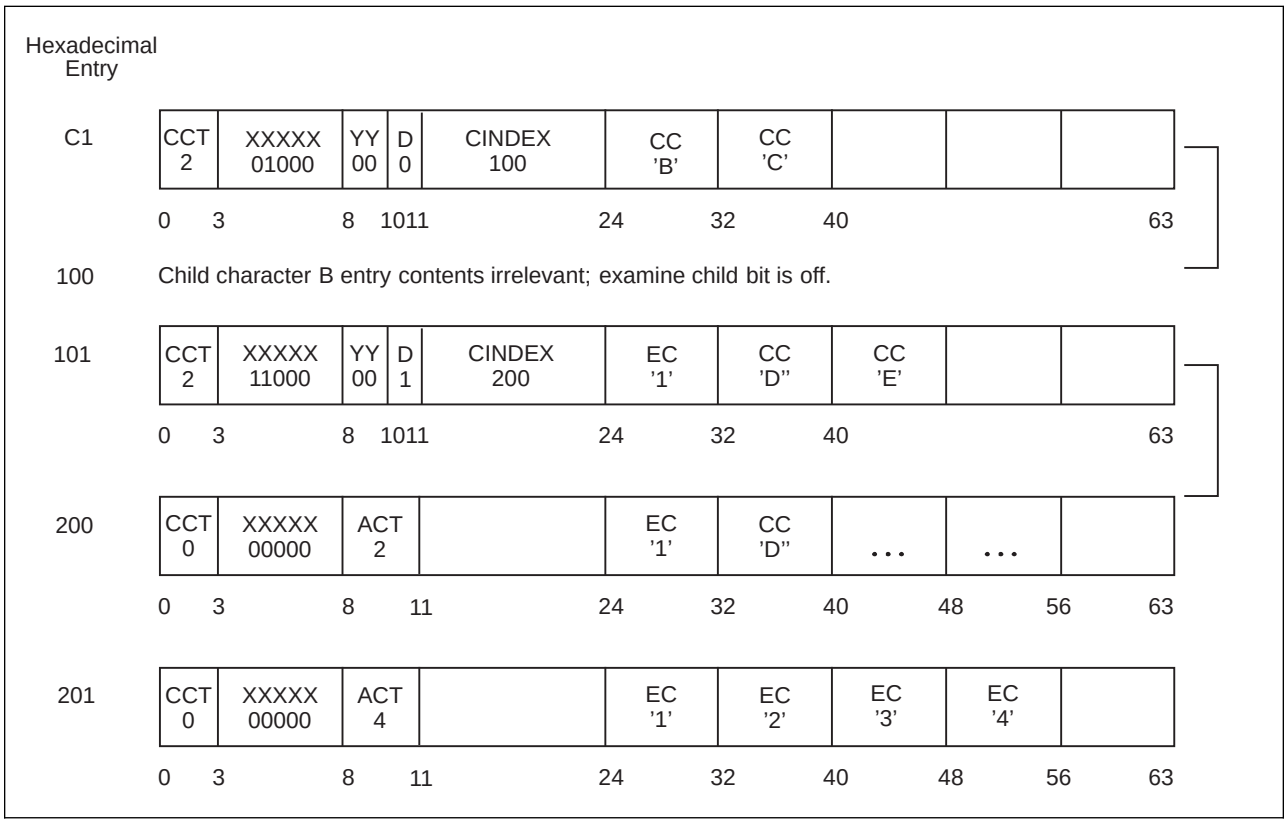
Example 1

Suppose the dictionary looks like the following:

Hexadecimal Entry	Description
C1	Alphabet entry for character A; 2 child characters B and C. The first child index is X'100'.
100	Entry for character B; no additional extension characters; no children.
101	Entry for character C; additional extension character 1; 2 child characters D and E. The first child index is X'200'.

200 Entry for character D; 2 additional extension characters 1 and 2; no children.

201 Entry for character E; 4 additional extension characters 1, 2, 3, and 4; no children.



If the input string is AD, the output string will consist of 2 compression symbols: one for A and one for D. When examining the dictionary entry for character A, the system determines that none of A's children match the next input character, D, and so returns the compression symbol for A. When examining the dictionary entry for character D, the system determines that it has no children, and so returns the compression symbol for D.

If the input string is AB, the output string will consist of 1 compression symbol for both input characters. When examining the dictionary input for character A, the system determines that A's first child character matches the next input character, B, and so looks at entry X'100'. Because that entry has no additional extension characters, a match is determined. Because there are no further input characters, the scan concludes.

If the input string is AC, the output string will consist of 2 compression symbols: one for A and one for C. When examining the dictionary input for character A, the system determines that A's second child character matches the next input character, C, and so looks at entry X'101'. Because that entry has an additional extension character, but the input string does not contain this character, no match is made, and the output is the compression symbol for A. Processing character C results in the compression symbol for C.

If the input string is AC1, the output string will consist of 1 compression symbol. When examining the dictionary input for character A, the system determines that A's second child character matches the next input character, C, and so looks at entry X'101'. Because that entry has an additional extension character, and the input string does contain this character, 1, a match is made, and the output is the compression symbol for AC1.

Similarly, the set of input strings longer than one character compressed by this dictionary are:

Hexadecimal Symbol	String
100	AB
101	AC1
200	AC1D12
201	AC1E1234

The compression symbol is the index of the dictionary entry. Based on this, you can see that the expansion dictionary must result in the reverse processing; for example, if a compression symbol of X'201' is found, the output must be the string AC1E1234. See “Expansion Dictionary Example” on page 22-20 for expansion dictionary processing.

Example 2 for More than 5 Children

Suppose the dictionary looks like the following:

Hexadecimal Entry	Description
C2	Alphabet entry for character B; child count of 6 (indicating 5 children plus a sibling descriptor); first child index is X'400', children are 1, 2, 3, 4, and 5.
400	Entry for character 1; no additional extension characters; no children.
401-404	Entries for characters 2 through 5; no additional extension characters; no children.
405	Sibling descriptor; child count of 15, which indicates 14 children plus another sibling descriptor; sibling characters A, B, C, D, E, and F.
405	Sibling descriptor extension. In the **expansion dictionary entry X'405'**, the sibling characters are G, H, I, J, K, L, M, and N.
406	Entry for character A; no additional extension characters; no children.
407-413	Entries for characters B through N; no additional extension characters; no children.
414	Next sibling descriptor; child count of 2; child characters O and P.
415	Entry for character O; no additional extension characters; no children.
416	Entry for character P; no additional extension characters; no children.

Hexadecimal Entry

C2

CCT 6	XXXXX 00000	YY 11	D 0	CINDEX 400	CC '1'	CC '2'	CC '3'	CC '4'	CC '5'

0 3 8 1011 24 32 40 63

400 Child character 1 entry contents irrelevant; examine child bit is
401 off.
402 Child character 2 entry contents irrelevant; examine child bit is
403 off.
404 Child character 3 entry contents irrelevant; examine child bit is

405

SCT 15	YYYYYYYYYYYY 111111111111	SC 'A'	SC 'B'	SC 'C'	SC 'D'	SC 'E'	SC 'F'

0 4 16 24 32 40 48 56 63

405E

SC 'G'	SC 'H'	SC 'I'	SC 'J'	SC 'K'	SC 'L'	SC 'M'	SC 'N'

0 4 8 16 24 32 40 48 56 63

406 Child character A entry contents irrelevant; examine child bit is off.
407 Child character B entry contents irrelevant; examine child bit is off.
408 Child character C entry contents irrelevant; examine child bit is off.
409 Child character C entry contents irrelevant; examine child bit is off.
40A Child character E entry contents irrelevant; examine child bit is off.
40B Child character F entry contents irrelevant; examine child bit is off.
40C Child character G entry contents irrelevant; examine child bit is off.
40D Child character H entry contents irrelevant; examine child bit is off.
40E Child character I entry contents irrelevant; examine child bit is off.
40F Child character K entry contents irrelevant; examine child bit is off.
410 Child character L entry contents irrelevant; examine child bit is off.
411 Child character M entry contents irrelevant; examine child bit is off.
412

414

SCT 2	YYYYYYYYYYYY 000000000000	SC 'O'	SC 'P'				

0 4 16 24 32 40 48 56 63

415 Child character O entry contents irrelevant; examine child bit is off.
416 Child character P entry contents irrelevant; examine child bit is off.

The set of input strings longer than one character compressed by this dictionary are:

Hexadecimal Symbol String

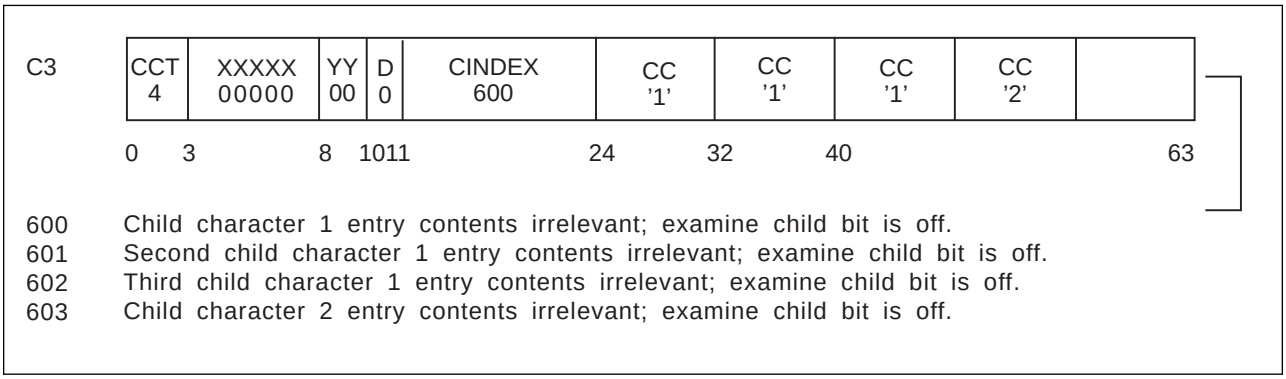
400-404	B1, B2, B3, B4, B5
406-40B	BA, BB, BC, BD, BE, BF
40C-413	BG, BH, BI, BJ, BK, BL, BM, BN
415-416	BO, BP

There are no compression symbols for 405 and 414. These are the sibling descriptor entries. Because their sibling descriptor extensions are located at those indices in the expansion dictionary (not the preceded or unpreceded entries required for expansion), it is important that no compression symbol have that value.

Example 3 for Children with the Same Value

Suppose the dictionary looks like the following:

Hexadecimal Entry	Description
C3	Alphabet entry for character C; child count of 4. The first child index is X'600' and the child characters are 1, 1, 1, and 2.
600	Entry for character 1; 4 additional extension characters A, B, C, and D; no children.
601	Entry for character 1; 3 additional extension characters A, B, and C; no children.
602	Entry for character 1; 2 additional extension characters A and B; no children.
603	Entry for character 2; no additional extension characters; no children.



The set of input strings longer than one character compressed by this dictionary are:

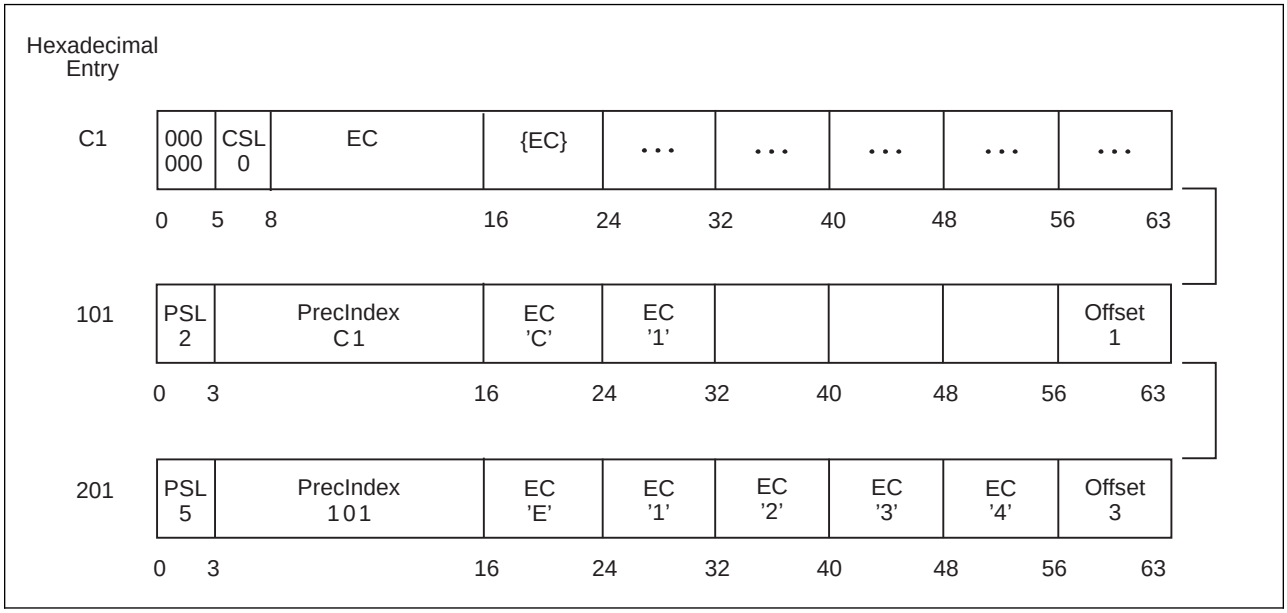
Hexadecimal Symbol String	
600	C1ABCD
601	C1ABC
602	C1AB
603	C2

By taking advantage of the special processing when the second and subsequent child characters match the first, you can reduce the number of dictionary entries searched to determine the compression symbols. For example, to find that X'601' is the compression symbol for the characters C1ABC, the processing examines entry X'C3', then entry X'600', then entry X'601'. Entry X'600' does not match because the input string does not have all 4 extension characters. There are alternate ways of setting up the dictionary to compress the same set of input strings handled by this dictionary.

Expansion Dictionary Example

Suppose the expansion dictionary looks like the following:

Hexadecimal Entry	Description
C1	Alphabet entry for character A. This by definition is an unpreceded entry.
101	A preceded entry, with characters C and 1; with preceding entry index of X'C1'; offset of 1.
201	A preceded entry, with characters E, 1, 2, 3, and 4; with preceding entry index of X'101'; offset of 3.



When processing an input compression symbol of X'201':

- Characters E1234 are placed at offset 3, and processing continues with entry X'101'.
- Characters C1 are placed at offset 1, and processing continues with entry X'C1'.
- Character A is placed at offset 0.

The expansion results in the 8 characters A, C, 1, E, 1, 2, 3, and 4 placed in the output string.

Chapter 23. 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. You can also use the UCBINFO macro to obtain device information from a UCB.

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.

© Copyright IBM Corp. 1988, 1999

- 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 with the COPY keyword to scan UCBs. On each invocation, UCBSCAN may return, in caller-supplied storage areas, a copy of one or more of the following UCB segments:

- UCB common segment
- UCB common extension segment
- UCB prefix extension segment
- UCB device class extension segment

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,        +
               RANGE=ALL,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 UCB Information for a Specified Device

You can use the UCBINFO macro to obtain information from a UCB for a specified device. You can use UCBINFO to obtain:

- A count of the UCBs for a device class
- Reasons why a device is offline
- Device path information
- Channel path type information
- A copy of the UCB prefix extension

- Information about the alias UCBs for a parallel access volume.

When you call UCBINFO, you provide the device number of the device for which you want the information.

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 *OS/390 HCD Planning* for further information on the EDT.

Using the EDTINFO Macro

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 below 16 megabyte UCBs for static and installation-static devices with 3-digit device numbers. However, you can use the RTNDEVN keyword with the EDTINFO macro to return a device number list for devices that are dynamic, 4-digit or described by UCBs residing above the 16-megabyte line.

  The UCBINFO macro, described earlier in this section, can then be used to obtain device number information for a specific device number.

  If your program is authorized, running in supervisor state or with a program key mask of 0-7, you can use the UCBLOOK macro to the

obtain the actual UCB address from a given device number. See, *OS/390 MVS Programming: Authorized Assembler Services Reference SET-WTO*, and *OS/390 MVS Programming: Authorized Assembler Services Guide* for the UCBLOOK macro.

- 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 the following be included in the list:
  - Devices defined to the system as dynamic
  - Devices defined with 4-digit device numbers
  - Devices with UCBs defined above 16 megabytes
- 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.

Chapter 24. The Internal Reader

The internal reader is a software substitute for a card punch and a card reader, a tape drive, or a TSO/E terminal. Instead of entering a job into the system (through JES) on punched cards, or through tape, you can use the **output** of one job or step as the **input** to another job, which JES will process directly.

The internal reader facility is useful for several kinds of applications:

- You can use it to generate another job or a series of jobs from an already-executing job. An online application program may submit another job to produce a report, for example, so it does not have to do it itself.
- A job that produces a series of jobs can put its output to an internal reader for immediate execution. For example, a job that updates data bases and starts other applications based upon some input parameters or real-time events, can use the internal reader for its output.
- The operator can start utility programs to read jobs from disk or tape files and submit them to the system. The IBM-supplied procedure 'RDR' is an example of a program that does this (see *OS/390 JES2 Initialization and Tuning Guide*).
- The operating system itself uses internal readers for submitting the JCL to start up started tasks or TSO/E logons.

Following is a discussion of the batch job internal reader, which is the facility you can use to submit a job from within another job.

Setting Up and Using an Internal Reader

The process of setting up and using an internal reader involves five tasks:

- Creating and allocating a data set
- Opening the data set
- Putting records into the data set
- Closing/deallocating the data set
- Passing the data set/records to the job entry subsystem for processing

Started tasks that run under the MSTR subsystem have the ability to set up an internal reader. To accomplish this they must first successfully invoke the Request Job ID SSI call, and then perform allocation. For more information on the Request Job ID call (SSI function code 20), see *OS/390 MVS Using the Subsystem Interface*.

Note: The VSAM interface and all data management macros and routines reside below 16Mb and operate in 24-bit addressing mode. Therefore, your internal reader data sets will be allocated storage below the 16Mb line. (See *OS/390 MVS Programming: Assembler Services Reference* for more information.)

© Copyright IBM Corp. 1988, 1999

Allocating the Internal Reader Data Set

You can allocate an internal reader data set, in any address space, either with JCL or dynamically, as follows:

- **Define the data set in the JCL for a job:**

  ```
  //JOBJCL DD SYSOUT=(A,INTRDR)
  ```

  Notes:

  1. "INTRDR" is an IBM-reserved name identifying the internal reader as the program to process this data set after it is created and written to.
  2. The SYSOUT class on this DD statement becomes the message class for the submitted job unless you specify MSGCLASS on the JOB statement.

- **Use the following dynamic allocation text unit keys to dynamically allocate an internal reader data set:**
  - DALSYSOU — define the SYSOUT data set and its class.
  - DALSPGNM — specify the SYSOUT program name (INTRDR).
  - DALCLOSE — request that INTRDR be deallocated at close.
  - DALRTDDN — request the return of the ddname assigned by dynamic allocation.
  - DALLRECL — specify the record length of any instream data set.
  - DALRECFM — specify the record format of any instream data set.

  Note: DALCLOSE, DALRTDDN, DALLRECL and DALRECFM are optional dynamic allocation text unit keys.

  For the format details of dynamic allocation text unit keys, see *OS/390 MVS Programming: Authorized Assembler Services Guide*.

Notes:

1. An INTRDR data set can contain any number of jobs.
2. The output destination of the INTRDR data set becomes the default print/punch routing of all jobs contained within it.
3. INTRDR data sets contain sequential, fixed-length, or variable-length records. Instream data records can be up to 254 bytes long. Note, however, that JCL images in the data sets must be exactly 80 bytes long.

Opening the Internal Reader Data Set

Note that if DALRTDDN was used, the returned DDNAME needs to be moved from the DALRTDDN text unit to the DCB before the OPEN macro is issued. Open the internal reader data set with the OPEN macro, specifying DSORG=PS. For information on using the OPEN macro, see *DFSMS/MVS Macro Instructions for Data Sets*.

Opening the INTRDR data set identifies it to the JES and prepares it to receive records.

Opening an internal reader that is already open has no effect. Multiple tasks cannot have the same internal reader open at the same time.

Sending Job Output to the Internal Reader

Code a WRITE (BSAM), PUT (QSAM) or PUT(VSAM) macro to send records to the internal reader.

Obtaining a Job Identifier

If you want to obtain the job identifier for a job, you must use VSAM data management macros. Issue an ENDREQ macro after writing a complete job to the internal reader. The job identifier is returned in the RPLRBAR field of the request parameter list (RPL). See *OS/390 JES2 Commands* or *OS/390 JES3 Commands* for details about the job identifier. RPLRBAR is an 8 byte field with a format of xxxnnnnn. The first 3 bytes, xxx, are the characters JOB, TSU or STC. The remaining 5 bytes, nnnnn, represent the five digits of the job number. See *OS/390 JES2 Initialization and Tuning Guide* or *OS/390 JES3 Initialization and Tuning Guide* for more information. If you submit JCL, and JES does not recognize it as a job, RPLRBAR contains blanks or a job id from an earlier job submitted through the internal reader. Note that the RPL cannot have records longer than 80 bytes. Specify the following options on the RPL macro when creating an RPL:

```
OPTCD=(ADR,SEQ,SYN,NUP),RECLEN=80
```

Where:

ADR Specifies addressed data processing with no index references.

SEQ Specifies sequential processing.

SYN Specifies a synchronous request and that control should be returned after completion of the request.

NUP Specifies non-update mode (records retrieved are not updated or deleted).

RECLEN=80
Specifies that the submitted JCL records are 80 bytes.

Note that you must issue a CLOSE macro after the last ENDREQ.

Closing the Internal Reader Data Set

While your program is writing records to the internal reader data set, the internal reader facility is writing them into a buffer in your address space. Issue the CLOSE macro to close the internal reader data set, and to send the contents of the buffer to JES.

Filling the buffer in your address space sends the records to the JES for processing. JES considers a closed data set to be a completed job stream and treats it as input. You can also send an internal reader data set to the JES for processing by coding one of the following:

1. Code /*EOF as the last record in the job.

   This control statement delimits the current job and makes it eligible for immediate processing by the JES2 input service. The internal reader data set remains open.

2. Code /*DEL as the last record in the job.

   This control statement cancels the job, and requests the output from the job. The job is immediately scheduled for output processing. The output will consist

of any JCL submitted so far, followed by a message indicating that the job has been deleted before execution.

3. Code /*PURGE as the last record in the job.

   This control statement is used only by JES2 internal readers. It cancels the current job and schedules it for purge processing; no output is generated for the job.

4. Code /*SCAN as the last record in the job.

   This statement also applies only to JES2 internal readers. It requests that the current job be scanned for JCL errors, but not executed.

You can put several groups of output records into the internal reader data set simply by starting each group with another JCL JOB statement. The following example illustrates this.

```
//JOBA     JOB   D58ELM1,MORRIS
//GENER    EXEC  PGM=IEBGENER
//SYSIN    DD    DUMMY
//SYSPRINT DD    SYSOUT=A,DEST=2NDFLOOR
//SYSUT2   DD    SYSOUT=(M,INTRDR)
//SYSUT1   DD    DATA
//JOBB     JOB   D58ELM1,MORRIS,MSGLEVEL=(1,1)
//REPORT1  EXEC  PGM=SUMMARY
//OUTPUT   DD    SYSOUT=*
//INPUT    DD    DSN=REPORTA,DISP=OLD
//JOBC     JOB   D58ELM1,MORRIS,MSGLEVEL=(1,1)
//REPORT2  EXEC  PGM=SUMMARY
//OUTPUT   DD    SYSOUT=A,DEST=3RDFLOOR
//INPUT    DD    DSN=REPORTB,DISP=OLD
/*EOF
```

In the preceding example, the IBM-supplied utility program IEBGENER is executed by job A. It reads from SYSUT1, and submits to the internal reader, jobs B and C, which are report-producing programs. Note that the message class for jobs B and C will be M, the SYSOUT class on the internal reader DD statement. Also, the OUTPUT data set from job B, because it specifies “*” (defaulting to the job's message class), will be class M.

The /*EOF control statement following the JCL indicates that the preceding jobs can be sent immediately to the job entry subsystem for input processing. Coding the CLOSE macro would have the same effect.

See *OS/390 JES2 Initialization and Tuning Guide* or *OS/390 JES3 Initialization and Tuning Guide* for more information about setting up and using internal readers.

Chapter 25. Using the Symbol Substitution Service

The ASASYMBM service substitutes text for symbols in character strings. This chapter explains how to explicitly call ASASYMBM to substitute text for symbols that you specify in application and vendor programs. It:

- Describes what symbols are
- Lists the symbols that the system provides
- Describes how to call ASASYMBM to substitute text for symbols.

Note: The system automatically substitutes text for symbols in dynamic allocations, parmlib members, system commands, and job control language (JCL). The system does not provide automatic support for other interfaces, such as application and vendor programs. Those interfaces must call ASASYMBM to perform symbolic substitution.

What Are Symbols?

Symbols are elements that allow the system to use the same source code for two or more unique instances of the same program. Symbols represent the variable information in a program. The program calls the ASASYMBM macro to substitute text for the symbols.

For example, suppose you define the following data set name, with the &TCASEID. symbol as the low-level qualifier:

```
TEST.&TCASEID.
```

Then suppose that two different instances of a program each call ASASYMBM, one with TEST001 as the substitution text for &TCASEID and the other with TEST002 as the substitution text. The resulting data set names are:

```
TEST.TEST001
TEST.TEST002
```

Notice that one data set definition produces two unique instances of data sets to be used in the programs.

Types of Symbols

There are two types of symbols that you can specify in application or vendor programs:

- **System symbols** are defined to the system at initialization. Your installation specifies substitution texts for system symbols or accepts their default values. When a program calls ASASYMBM, it can accept the installation-defined substitution texts or override them. There are two types of system symbols:
  - *Static* system symbols have substitution texts that are defined at system initialization and remain fixed for the life of an IPL. Static system symbols represent fixed values such as system names and sysplex names.

    The DISPLAY SYMBOLS command displays the static system symbols and associated substitution texts that are currently in effect. See *OS/390 MVS System Commands* or *OS/390 MVS System Commands* for details about DISPLAY SYMBOLS.

© Copyright IBM Corp. 1988, 1999

– *Dynamic* system symbols have substitution texts that can change at any point in an IPL. They represent values that can change often, such as dates and times. A set of dynamic system symbols is defined to the system; your installation cannot provide additional dynamic system symbols.

See the section on using system symbols in *OS/390 MVS Initialization and Tuning Reference* for lists of dynamic and static system symbols.

- **User symbols** are symbols that a caller defines on a call to ASASYMBM. They are valid only for the specific call. The caller specifies the names and substitution texts for user symbols in a *symbol table*. If the names of user symbols are the same as the names of system symbols, the substitution texts for the user symbols override the substitution texts for the system symbols.

If your program accepts the substitution texts for the installation-defined system symbols, there is no need to specify those system symbols in the symbol table that you provide to ASASYMBM; substitution for those system symbols is performed automatically. However, if your program wants to override the installation-defined substitution texts for system symbols, you must specify those system symbols in the symbol table. The symbol table is the *only* place where you can specify user symbols.

Examples of User Symbols

Like system symbols, user symbols can represent any type of variable information in a program. When planning to define user symbols, you should first determine if the system symbols provided by the system, and their associated substitution texts, meet your needs. Define user symbols only if you need additional values.

Suppose you are writing a program that is to access several data sets each time it runs. The user is to provide the name of the specific data set to be used.

Your first step is to create a pattern, or “skeleton,” for the data set name. You decide to use the name of the system on which the program runs as a high-level qualifier, and the name of the data set that the user provides as a low-level qualifier.

You decide to use the &SYSNAME system symbol to identify the system on which the program runs. Because &SYSNAME is already defined to the system, and you do not want to override its substitution text, you do not need to provide it to ASASYMBM. Because the system does not define a symbol to identify the input from the user, you provide the following user symbol to ASASYMBM:

```
TESTDATA       DC    C'&&DATAID.'      Define the symbol &DATAID
```

You begin by specifying, in your program, references to three data sets:

```
&SYSNAME..&DATAID..DS1
&SYSNAME..&DATAID..DS2
&SYSNAME..&DATAID..DS3
```

You then specify that the user is to provide, as input to the program, the substitution text for the &DATAID user symbol. For example:

```
EXEC PGM=MYPGM,PARM='DATA1'
```

Your program provides, as input to ASASYMBM, a symbol table that contains the &DATAID user symbol, with the substitution text that the user provided as input to the program:

```
DATAIDSUBTEXT  DC    C'DATA1'          Substitution text for &DATAID
```

To determine the data set name to be used, your program also provides to ASAYSMBM the pattern for the data set name, including the symbols for which substitution is to occur. For example:

```
SYS1.&SYSNAME..&DATAID..DS1
SYS1.&SYSNAME..&DATAID..DS2
SYS1.&SYSNAME..&DATAID..DS3
```

The data set names resolve to the following names after symbolic substitution:

```
SYS1.DATA1.DS1
SYS1.DATA1.DS2
SYS1.DATA1.DS3
```

The following section explains how to call ASASYMBM to perform the substitution described in the example above.

Calling the ASASYMBM Service

A call to ASASYMBM:

1. Defines, to the system, the system symbols that ASASYMBM is to use.
2. Defines, to the system, *user symbols* that are specified in the symbol table (as described later).
3. Substitutes values for the symbols in the symbol table, based on an input character string (pattern).
4. Places the results of the substitution in an output buffer that the caller specifies.
5. Places the length of the output buffer in a field.
6. Provides a return code to the calling program.

You must include the ASASYMBP mapping macro to build the user parameter area (SYMBP) for ASASYMBM. The following sections describe how to set up ASASYMBP to enable the desired functions.

Setting Up the ASASYMBP Mapping Macro

Before calling ASASYMBM to substitute text for a symbol, the caller must provide the symbol pattern and its length, an output buffer and its length, and an area in which to place the return code from ASASYMBM. The caller can optionally provide a symbol table and a timestamp.

The caller *must* code the following fields in the ASASYMBP mapping macro:

Field	Description
SYMBPPATTERN@	Specifies the address of the input character string (pattern) that contains symbols to be resolved.
SYMBPPATTERNLENGTH	Specifies the length of the input pattern specified in the PATTERN@ field.

SYMBPTARGET@	Specifies the address of the buffer that is to contain the output from ASASYMBM (the results of the symbolic substitution).
SYMBPTARGETLENGTH@	Specifies the address of a fullword that: • On input, contains the length of the output buffer specified in the TARGET@ field • On output, contains the length of the substituted text within the output buffer.
SYMBPRETURNCODE@	Specifies the address of a fullword that is to contain the return code from ASASYMBM.

Before calling ASASYMBM, you can *optionally* code the following fields in the ASASYMBP mapping macro:

Field	Description
SYMBPSYMBOLTABLE@	Specifies the address of a symbol table mapped by the SYMBT DSECT. Specify an address in this field if you want to do one or both of the following: • Provide symbols in addition to the system symbols that are defined to the system, or override the system symbols that are defined to the system. • Use additional functions of ASASYMBM, which are described in "Providing a Symbol Table to ASASYMBM." Otherwise, specify an address of 0 in this field.
SYMBPTIMESTAMP@	Specifies the address of an eight-character area that contains the time stamp to be used. Specify an address of zero to have the system use the current timestamp when substituting for system symbols that relate to the time or date.

After the call to ASASYMBM, the caller can examine the following:

- The fullword pointed to by the SYMBPRETURNCODE@ field
- The fullword pointed to by the SYMBPTARGETLENGTH@ field
- The area pointed to by the SYMBPTARGET@ field.

Providing a Symbol Table to ASASYMBM

A program that calls ASASYMBM can optionally provide user symbols and their associated substitution texts in a *symbol table*. The SYMBPSYMBOLTABLE@ field, in the user parameter area of the ASASYMBP mapping macro, specifies the address of a symbol table, which is mapped by the SYMBT DSECT.

Setting Up the Symbol Table

When you provide a symbol table to the ASASYMBM service (that is, when you specify a non-zero address in the SYMBPSYMBOLTABLE@ field), code the following fields in the ASASYMBP mapping macro:

Field	Description
SYMBTNUMBEROFSYMBOLS	Specifies the number of entries in the symbol table. The number can be zero.
SYMBTTABLEENTRIES	Specifies the beginning of the entries in the symbol table. If the SYMBTINDIRECTSYMBOLAREA bit (described later in this section) is off, the symbol table entries must be contiguous to the header, beginning at this field. Otherwise, place the SYMBTESYMBOLAREAADDR field in this field; SYMBTESYMBOLAREAADDR must point to an area that contains the symbol table entries. SYMBTESYMBOLAREAADDR is a field in the SYMBTE structure; see the ASASYMBP mapping in *OS/390 MVS Data Areas, Vol 1 (ABEP-DALT)* for details.
SYMBTE	Specifies an entry in the symbol table. Code one symbol table entry (SYMBTE) for each symbol in the symbol table. The number of entries is specified in the SYMBTNUMBEROFSYMBOLS field. If the SYMBTINDIRECTSYMBOLAREA bit is off, the symbol table entries must be contiguous to the header, beginning at the SYMBTTABLEENTRIES field. Otherwise, the symbol table entries begin at the area pointed to by the SYMBTESYMBOLAREAADDR field. Each symbol table entry (SYMBTE) must set the following fields: • SYMBTESYMBOLPTR, when the SYMBTPTRSAREOFFSETS flag is **off**; it specifies the address of the area that contains the name of the symbol • SYMBTESYMBOFFSET, when the SYMBTPTRSAREOFFSETS flag is **on**; it specifies the offset to the symbol from the beginning of the symbol area; you must also set up the area pointed to by the offset • SYMBTESYMBOLLENGTH, which specifies the length of the symbol • SYMBTESUBTEXTPTR, when the SYMBTPTRSAREOFFSETS flag is **off**; it specifies the address of the area that contains the substitution text for the symbol • SYMBTESUBTEXTOFFSET, when the SYMBTPTRSAREOFFSETS flag is **on**; it

specifies the offset to the substitution text from the beginning of the symbol area; you must also set up the area pointed to by the offset

- SYMBTESUBTEXTLENGTH, which specifies the length of the substitution text.

If you specify an address in the SYMBPSYMBOLTABLE@ field, you must also provide a symbol table header (SYMBTHEADER). Optionally set one or more of the flags in the symbol table header, listed in this section, to indicate that ASASYMBM is to perform additional functions.

Note: The field and bit names in the following section are all prefixed by the characters SYMBT in the ASASYMBP mapping macro. The SYMBT prefix is not used in this section out of consideration for readability.

Flag	Function
CHECKNULLSUBTEXT	Specifies that ASASYMBM is to return a return code of X'0C' if the substitution text for a symbol is a null string (has a length of zero). The default is to *not* check if the substitution text for a symbol is a null string.
INDIRECTSYMBOLAREA	Specifies that the symbol area is not contiguous; it is pointed to by the SYMBTESYMBOLAREAADDR field. For an example of a symbol area that is not contiguous, see Figure 25-2 on page 25-9. The default is that the symbol area is contiguous.
NODEFAULTSYMBOLS	Specifies that ASASYMBM is not to use the default set of system symbols. In other words, ASASYMBM uses only the symbols that are defined in the user-provided symbol table. When you *do not* set this flag, ONLYDYNAMICSYMBOLS, or ONLYSTATICSYMBOLS to **on**, the default is to use both dynamic and static system symbols.
ONLYDYNAMICSYMBOLS	Specifies that ASASYMBM is not to substitute text for the *static* system symbols.
ONLYSTATICSYMBOLS	Specifies that ASASYMBM is not to substitute text for the *dynamic* system symbols.
PTRSAREOFFSETS	Specifies that the pointer fields in the symbol table are offsets. The system adds the offset to the address of the symbol area to obtain the actual address of the operand. The default is to indicate that the pointer fields in the symbol table are pointers.
MIXEDCASESYMBOLS	Specifies that the system is to recognize, within an input pattern, symbols that contain: • All upper-case characters, and • Both upper-case and lower-case characters.

The default is to recognize symbols that match the symbols in the symbol table.

TIMESTAMPISGMT Specifies that the input time stamp is Greenwich Mean Time (GMT). TIMESTAMPISGMT is the default when you provide a timestamp (in the area pointed to by SYMBPTIMESTAMP@) and you do not set this flag, TIMESTAMPISLOCAL, or TIMESTAMPISSTCK to **on**.

TIMESTAMPISLOCAL Specifies that the input timestamp is local time.

TIMESTAMPISSTCK Specifies that the input timestamp is obtained from the system time of day (TOD) clock.

WARNNOSUB Specifies that ASASYMBM is to return a X'10' return code when the system does not perform symbolic substitution.

The default is to *not* return a return code when the system does not perform symbolic substitution.

WARNSUBSTRINGS Specifies that ASASYMBM is to return a X'04' return code when the system finds a substring error. See the section on using symbols in *OS/390 MVS Initialization and Tuning Reference* for information about how the system performs symbolic substitution when errors occur in substringing.

The default is to *not* return a return code when the system finds a substring error.

The following is an example of a symbol table that is contiguous:

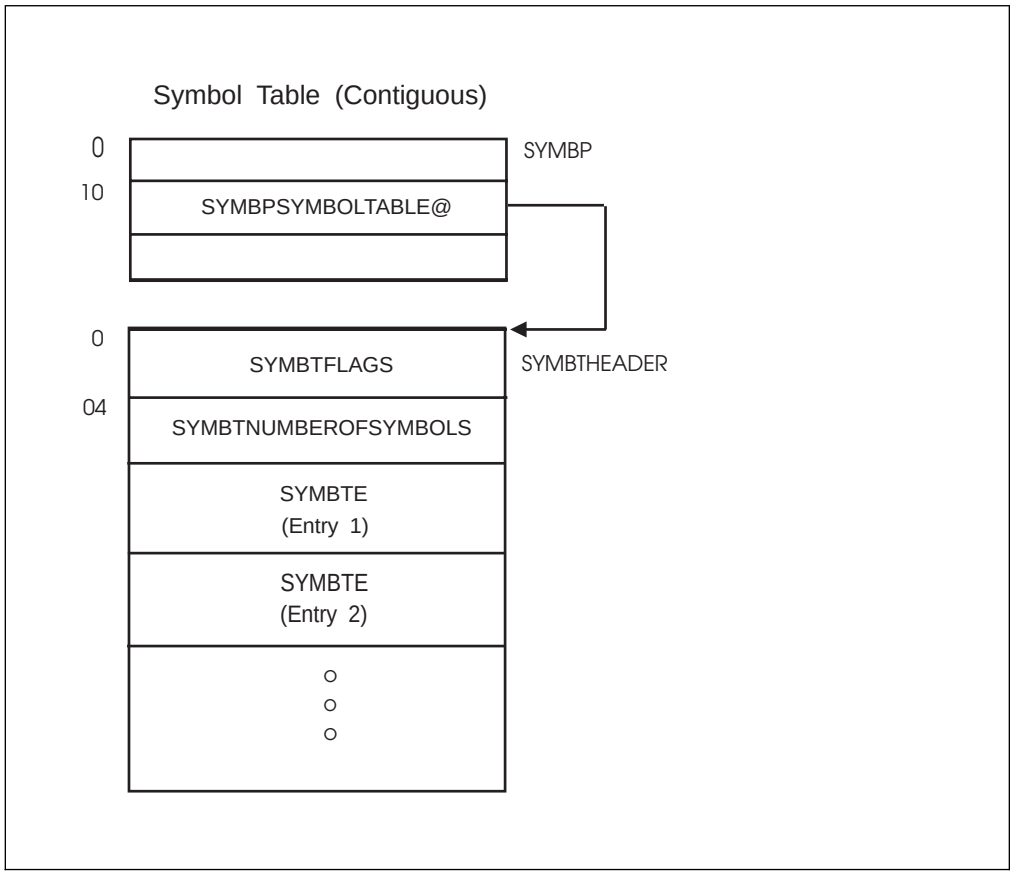


Figure 25-1. Contiguous Symbol Table

The following is an example of a symbol table that is not contiguous:

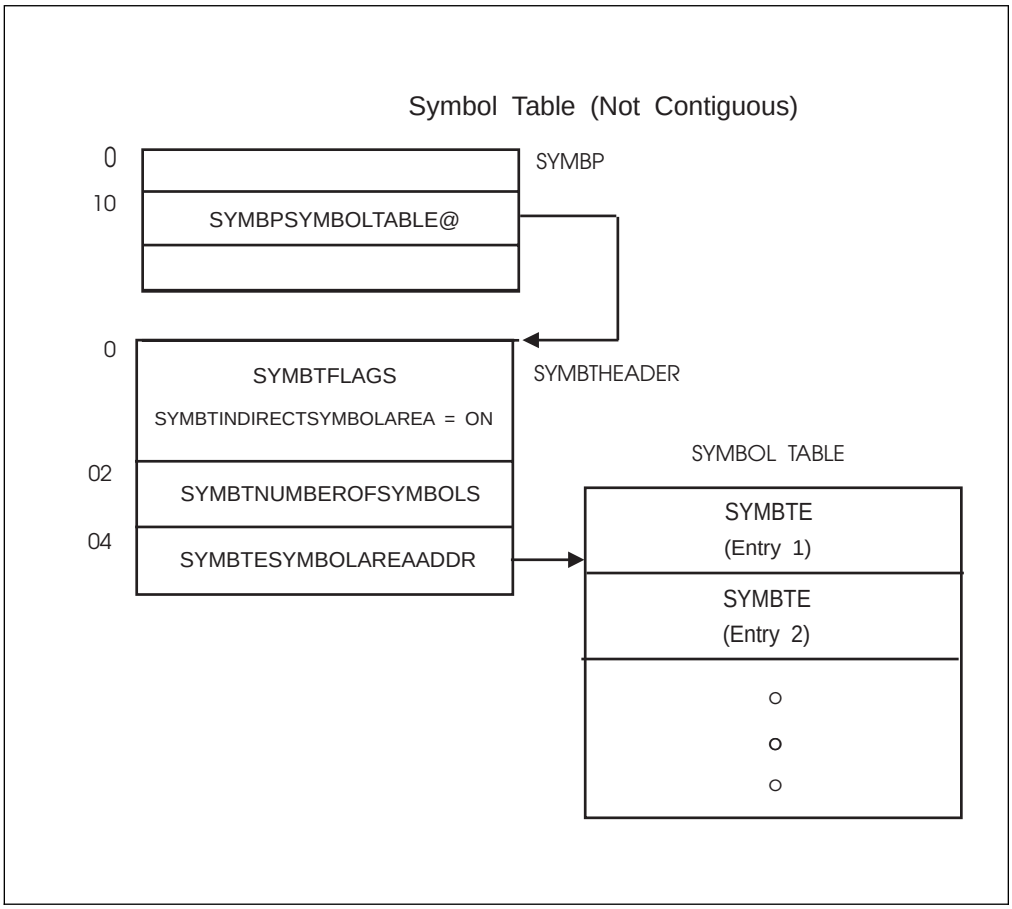


Figure 25-2. Non-contiguous Symbol Table

Rules for Entering Symbols in the Symbol Table

Follow these rules when specifying entries in the symbol table:

1. The names of the symbols must:
   - Begin with an ampersand (&)
   - End with a period (.)
   - Contain 1-253 *additional* characters (in other words, you cannot create the symbol "&.").

   Note: It is important that all symbols end with a period. If a symbol does not end with a period, syntax errors could result.
2. Specify the names of system symbols only if you want to override the installation-defined substitution texts for those system symbols. Ask the operator to enter a DISPLAY SYMBOLS command to display the system symbols and associated substitution texts that are currently in effect.
3. Do not begin the names of user symbols with the characters &SYS. The &SYS prefix is reserved for system symbols; use the &SYS prefix only when overriding substitution texts for system symbols.
4. The length of the substitution text cannot exceed the length of the symbol name *plus* the ampersand (&).

This restriction prevents the flow of characters beyond established limits for text entry. For example, assume that the limit for text entry is column 71. If a line that contains a four-character symbol extends to column 70, a substitution text of greater than six characters would force the text beyond column 71.

Using Symbols in Programs

Your program can call ASASYMBM to substitute text for the symbols that are specified in the program. The following examples show how to call ASASYMBM in various situations.

Example 1

Operation: Set up the area that is to contain the substitution text. The caller does not provide a symbol table or timestamp.

```
        LA    3,MYSYMBP
        USING SYMBP,3
        XC    SYMBP(SYMBP_LEN),SYMBP   Initialize to zero
        LA    4,PATTERN                Address of pattern
        ST    4,SYMBPPATTERN@          Save in SYMBP area
        LA    4,L'PATTERN              Length of pattern
        ST    4,SYMBPPATTERNLENGTH     Save in SYMBP area
        LA    4,TARGET                 Address of target
        ST    4,SYMBPTARGET@           Save in SYMBP area
        MVC   TARGETLENGTH,=A(L'TARGET)  Set length of target
        LA    4,TARGETLENGTH           Address of target length
        ST    4,SYMBPTARGETLENGTH@     Save in SYMBP area
*
* Because the caller did not provide a symbol table, we know that
* we are using only the system symbols provided by MVS.  Since we have
* initialized the entire SYMBP area to 0, we do not have to
* set up the SYMBPSYMBOLTABLE@ field.
*
* Because the caller did not provide a timestamp, and because we
* have initialized the entire SYMBP area to 0, we do not have to
* set up the SYMBPTIMESTAMP@ field.
*
        LA    4,RETURNCODE             Address of return code
        ST    4,SYMBPRETURNCODE@       Save in SYMBP area
        DROP  3
⋮
*
* Note that to avoid the assembler substituting
* for &SYSNAME, &YYMMDD, &HHMMSS, two ampersands are specified.
* The resulting pattern, then, is actually
*      USERID.&SYSNAME..D&YYMMDD..T&HHMMSS
*
PATTERN      DC    C'USERID.&&SYSNAME..D&&YYMMDD..T&&HHMMSS'
DYNAREA      DSECT
             DS    0F
MYSYMBP      DS    CL(SYMBP_LEN)       SYMBP area
RETURNCODE   DS    F                   Return code
TARGETLENGTH DS    F                   Length of target
TARGET       DS    CL80 An area big enough to hold the target no
*                       matter what is substituted. Since &DATE
*                       and &TIME are not used, it need be no
*                       longer than the pattern area.
             ASASYMBP   ,              Mapping of SYMBP area
```

Example 2

Operation: Set up the SYMBP area. Provide a symbol table and a timestamp.

```
        LA    3,MYSYMBP
        USING SYMBP,3
        XC    SYMBP(SYMBP_LEN),SYMBP   Initialize to zero
        LA    4,PATTERN                Address of pattern
        ST    4,SYMBPPATTERN@          Save in SYMBP area
        LA    4,L'PATTERN              Length of pattern
        ST    4,SYMBPPATTERNLENGTH     Save in SYMBP area
        LA    4,TARGET                 Address of target
        ST    4,SYMBPTARGET@           Save in SYMBP area
        MVC   TARGETLENGTH,=A(L'TARGET)  Set length of target
        LA    4,TARGETLENGTH           Address of target length
        ST    4,SYMBPTARGETLENGTH@     Save in SYMBP area
        LA    5,MYSYMBT                Address of symbol table
        ST    5,SYMBPSYMBOLTABLE@      Save in SYMBP area
*
* Initialize symbol table to indicate that the input timestamp
* is local time, not GMT, and to contain two system symbols.
*
        USING SYMBT,5
        XC    SYMBTHEADER,SYMBTHEADER  Clear symbol table header
        OI    SYMBTFLAGS,SYMBTTIMESTAMPISLOCAL  Local timestamp
        LA    6,2                      Number of symbols
        STH   6,SYMBTNUMBEROFSYMBOLS   Save in SYMBT area
        LA    5,SYMBTTABLEENTRIES      Address of first symbol entry
        USING SYMBTE,5                 A symbol table entry
*
* Initialize first entry in symbol table.
*
        LA    6,SYMBOL1                Address of first symbol
        ST    6,SYMBTESYMBOLPTR        Save in SYMBTE area
        LA    6,L'SYMBOL1              Length of first symbol
        ST    6,SYMBTESYMBOLLENGTH     Save in SYMBTE area
        LA    6,SYMBOL1SUBTEXT         Address of substitution text
        ST    6,SYMBTESUBTEXTPTR       Save in SYMBTE area
        LA    6,L'SYMBOL1SUBTEXT       Length of substitution text
        ST    6,SYMBTESUBTEXTLENGTH    Save in SYMBTE area
*
* Move to next entry in symbol table.
*
        LA    5,SYMBTE_LEN(,5)         Address of next symbol entry
*
* Initialize second entry in symbol table.
*
        LA    6,SYMBOL2                Address of symbol
        ST    6,SYMBTESYMBOLPTR        Save in SYMBTE area
        LA    6,L'SYMBOL2              Length of symbol
        ST    6,SYMBTESYMBOLLENGTH     Save in SYMBTE area
        LA    6,SYMBOL2SUBTEXT         Address of substitution text
        ST    6,SYMBTESUBTEXTPTR       Save in SYMBTE area
        LA    6,L'SYMBOL2SUBTEXT       Length of substitution text
        ST    6,SYMBTESUBTEXTLENGTH    Save in SYMBTE area
        DROP  5                        No longer need addressability
```

```
*
* Complete parameter area initialization.
*
        LA    4,MYTIMESTAMP            Address of timestamp
        ST    4,SYMBPTIMESTAMP@        Save in SYMBP area
        LA    4,RETURNCODE             Address of return code
        ST    4,SYMBPRETURNCODE@       Save in SYMBP area
        DROP  3                        No longer need addressability
⋮
*
* Note that in order to avoid the assembler's substituting
* for &YYMMDD, &HHMMSS, &S1, &S2, two ampersands are specified.
* The resulting pattern, then, is actually
*      USERID.&YYMMDD..&S1..&S2
* Similarly, the resulting symbol names are
*      &S1; and &S2;
*
PATTERN        DC    C'USERID.D&&YYMMDD..T&&HHMMSS..&&S1..&&S2'
SYMBOL1        DC    C'&&S1.'          First symbol is &S1
SYMBOL1SUBTEXT DC    C'S1V'            Substitution text for &S1
SYMBOL2        DC    C'&&S2.'          Second symbol is &S2
SYMBOL2SUBTEXT DC    C'S2V'            Substitution text for &S2
* Note that the substitution text values above are no longer than
* the symbol names (counting the "&" but not the "."). This
* helps to ensure that the substituted length is not greater
* than the pre-substitution length.
*
DYNAREA        DSECT
               DS    0F
MYSYMBP        DS    CL(SYMBP_LEN)     SYMBP area
               DS    0F
MYSYMBT        DS    CL(SYMBT_LEN+2*SYMBTE_LEN) Symbol table with
*                                        room for two symbol entries
MYTIMESTAMP    DS    CL8 Time stamp that was set previously
*                        Assume it represents local time
RETURNCODE     DS    F                 Return code
TARGETLENGTH   DS    F                 Input/Output Target Length
TARGET         DS    CL80  An area big enough to hold the target no
*                           matter what is substituted
               ASASYMBP   ,            Mapping of SYMBP, SYMBT, SYMBTE
```

Example 3

Operation: Use the LINK macro to invoke the ASASYMBM service:

```
* Set up MYSYMBP as in previous examples.
⋮
        LINK  EP=ASASYMBM,MF=(E,MYSYMBP)
⋮
DYNAREA DSECT
        DS    0F
MYSYMBP DS    CL(SYMBP_LEN)            SYMBP area
        ASASYMBP   ,                   Mapping of SYMBP area
```

Example 4

Operation: Use the LINKX macro to invoke the ASASYMBM service:

```
* Set up MYSYMBP as in previous examples.
⋮
        MVC   MYLIST(MYSLIST_LEN),MYSLIST Initialize execute form
        LINKX EP=ASASYMBM,MF=(E,MYSYMBP),SF=(E,MYLIST)  call service
⋮
MYSLIST LINKX SF=L                      Initialized list form
MYSLIST_LEN   EQU *-MYSLIST             Length of list form
DYNAREA DSECT
MYLIST  LINKX SF=L                      List form in dynamic area
        DS    0F
MYSYMBP DS    CL(SYMBP_LEN)             SYMBP area
        ASASYMBP   ,                    Mapping of SYMBP area
```

Chapter 26. Using System Logger Services

This chapter covers the information you need to write a system logger application, including the following topics:

- "What is System Logger?"
- "The System Logger Configuration" on page 26-4.
- "Overview of System Logger Services" on page 26-10.
- "IXGINVNT: Managing the LOGR Policy" on page 26-20.
- "IXGCONN: Connecting to and Disconnecting From a Log Stream" on page 26-22.
- "IXGWRITE: Writing to a Log Stream" on page 26-32.
- "IXGBRWSE: Browsing/Reading a Log Stream" on page 26-34.
- "IXGDELET: Deleting Log Blocks from a Log Stream" on page 26-38.
- "IXGIMPRT: Import Log Blocks" on page 26-39.
- "IXGQUERY: Get Information About a Log Stream" on page 26-41.
- "IXGOFFLD: Initiate Offload to DASD Log Data Sets" on page 26-45.
- "IXGUPDAT: Modify Log Stream Control Information" on page 26-46.
- "Setting Up the System Logger Configuration" on page 26-47.
- "Reading Data From Log Streams in Data Set Format" on page 26-49.
- "When Things Go Wrong - Recovery Scenarios for System Logger" on page 26-53.

What is System Logger?

System logger is a set of services that allows an application to write, browse, and delete log data. You can use system logger services to merge data from multiple instances of an application, including merging data from different systems across a sysplex.

For example, suppose you are concurrently running multiple instances of an application in a sysplex, and each application instance can update a common database. It is important for your installation to maintain a common log of all updates to the database from across the sysplex, so that if the database should be damaged, it can be restored from the backup copy. You can merge the log data from applications across the sysplex into a **log stream**, which is simply a collection of data in **log blocks** residing in the coupling facility and on DASD (see Figure 26-1 on page 26-2).

© Copyright IBM Corp. 1988, 1999

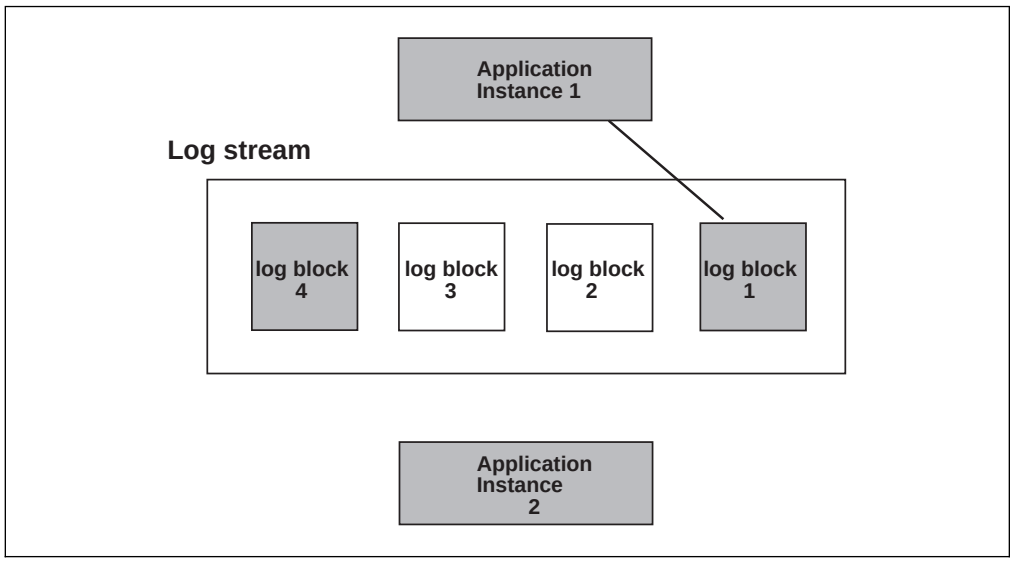


Figure 26-1. System Logger Log Stream

The Log Stream

A log stream is an application specific collection of data that is used as a log. The data is written to and read from the log stream by one or more instances of the application associated with the log stream. A log stream can be used for such purposes as a transaction log, a log for re-creating databases, a recovery log, or other logs needed by applications.

A system logger application can write log data into a **log stream**, which is simply a collection of data. Data in a log stream spans two kinds of storage:

- **Interim storage**, where data can be accessed quickly without incurring DASD I/O.
- **DASD log data set storage**, where data is hardened for longer term access. When the interim storage medium for a log stream reaches a user-defined threshold, the log data is offloaded to DASD log data sets.

There are two types of log streams; coupling facility log streams and DASD-only log streams. The main difference between the two types of log streams is the storage medium system logger uses to hold interim log data:

- In a coupling facility log stream, interim storage for log data is in coupling facility list structures. See “Coupling Facility Log Stream” on page 26-3.
- In a DASD-only log stream interim storage for log data is contained in local storage buffers on the system. **Local storage buffers** are data space areas associated with the system logger address space, IXGLOGR. See “DASD-Only Log Stream” on page 26-3.

Your installation can use just coupling facility log streams, just DASD-only log streams, or a combination of both types of log streams. The requirements and preparation steps for the two types of log streams are somewhat different; see “Setting Up the System Logger Configuration” on page 26-47.

Coupling Facility Log Stream

Figure 26-2 shows how a coupling facility log stream spans two levels of storage; the coupling facility for interim storage and DASD log data sets for more permanent storage. When the coupling facility space for the log stream fills, the data is offloaded to DASD log data sets. A coupling facility log stream can contain data from multiple systems, allowing a system logger application to merge data from systems across the sysplex.

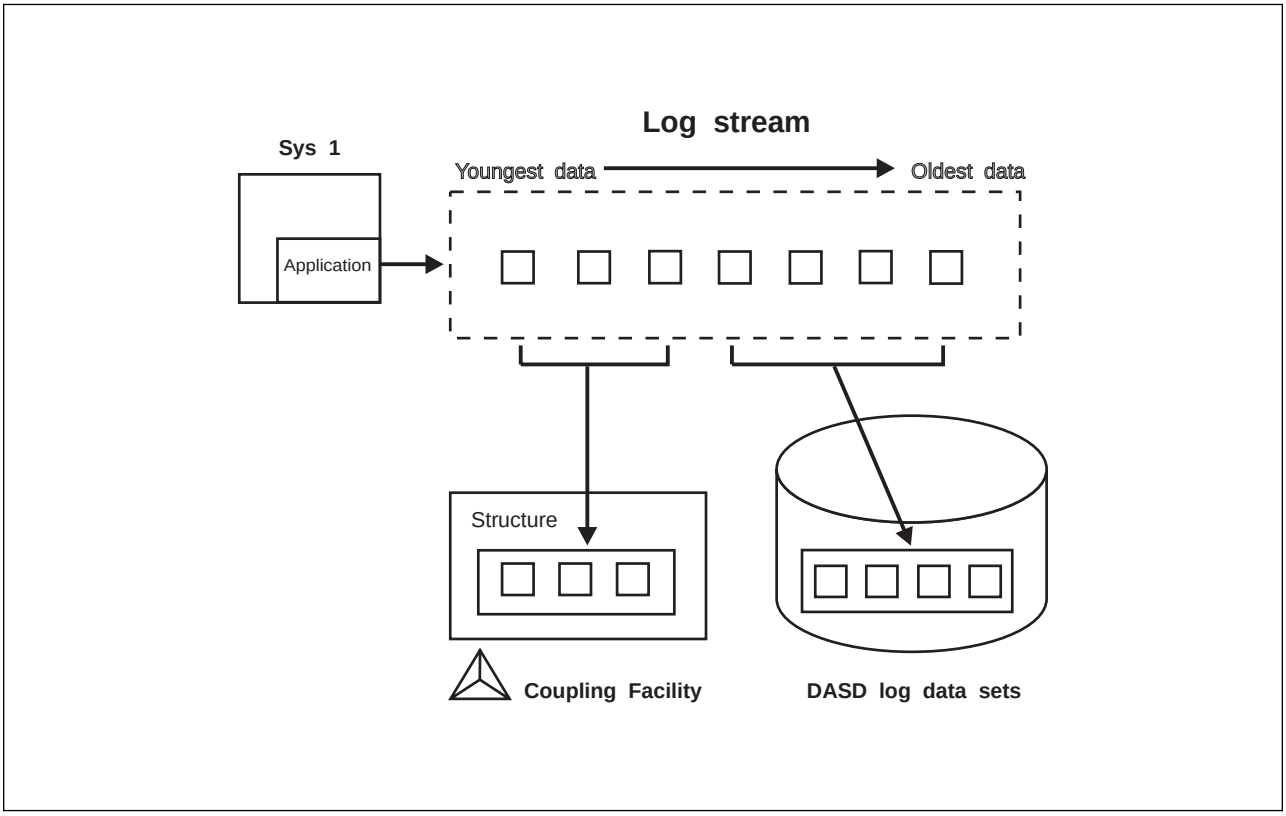


Figure 26-2. Log Stream Data on the Coupling Facility and DASD

When a system logger application writes a log block to a coupling facility log stream, system logger writes it first to a coupling facility list structure. System logger requires that a coupling facility list structure be associated with each log stream. When the coupling facility structure space allocated for the log stream reaches the installation-defined threshold, system logger moves (**offloads**) the log blocks from the coupling facility structure to VSAM linear DASD data sets, so that the coupling facility space for the log stream can be used to hold new log blocks. From a user's point of view, the actual location of the log data in the log stream is transparent.

DASD-Only Log Stream

Figure 26-3 on page 26-4 shows a DASD-only log stream spanning two levels of storage; local storage buffers for interim storage, which is then offloaded to DASD log data sets for more permanent storage.

A DASD-only log stream has a single-system scope; only one system at a time can connect to DASD-only log stream. Multiple applications from the same system can, however, simultaneously connect to a DASD-only log stream.

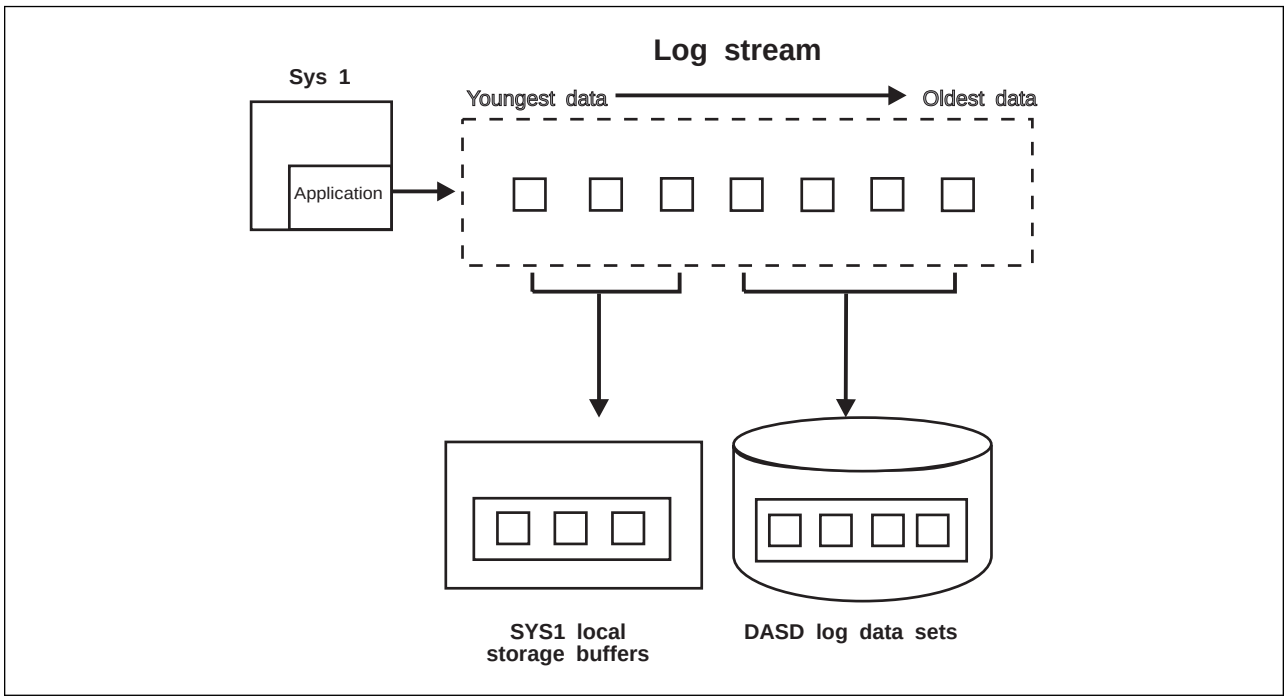


Figure 26-3. Log Stream Data in Local Storage Buffers and DASD Log Data Sets

When a system logger application writes a log block to a DASD-only log stream, system logger writes it first to the local storage buffers for the system and duplexes it to a DASD staging data set associated with the log stream. When the staging data set space allocated for the log stream reaches the installation-defined threshold, system logger offloads the log blocks from local storage buffers to VSAM linear DASD data sets. From a user's point of view, the actual location of the log data in the log stream is transparent.

Both a DASD-only log stream and a coupling facility log stream can have data in multiple DASD log data sets; as a log stream fills log data sets on DASD, system logger automatically allocates new ones for the log stream.

The System Logger Configuration

The system logger configuration you use depends on whether or not you use a coupling facility.

Coupling Facility Log Stream Configuration: Figure 26-4 on page 26-5 shows all the parts involved when a system logger application writes to a coupling facility log stream. In this example, a system logger application runs on two systems in a sysplex. Both instances of the application write data to the same log stream, TRANSLOG. Each system contains a system logger address space. A system logger application uses system logger services to access the system logger capabilities.

When a system logger application writes data to a coupling facility log stream, system logger writes the data to a coupling facility list structure associated with the log stream. Then, when the coupling facility structure fills with data, system logger offloads the data to DASD log data sets.

You can optionally elect to have coupling facility data duplexed to DASD staging data sets for a coupling facility log stream.

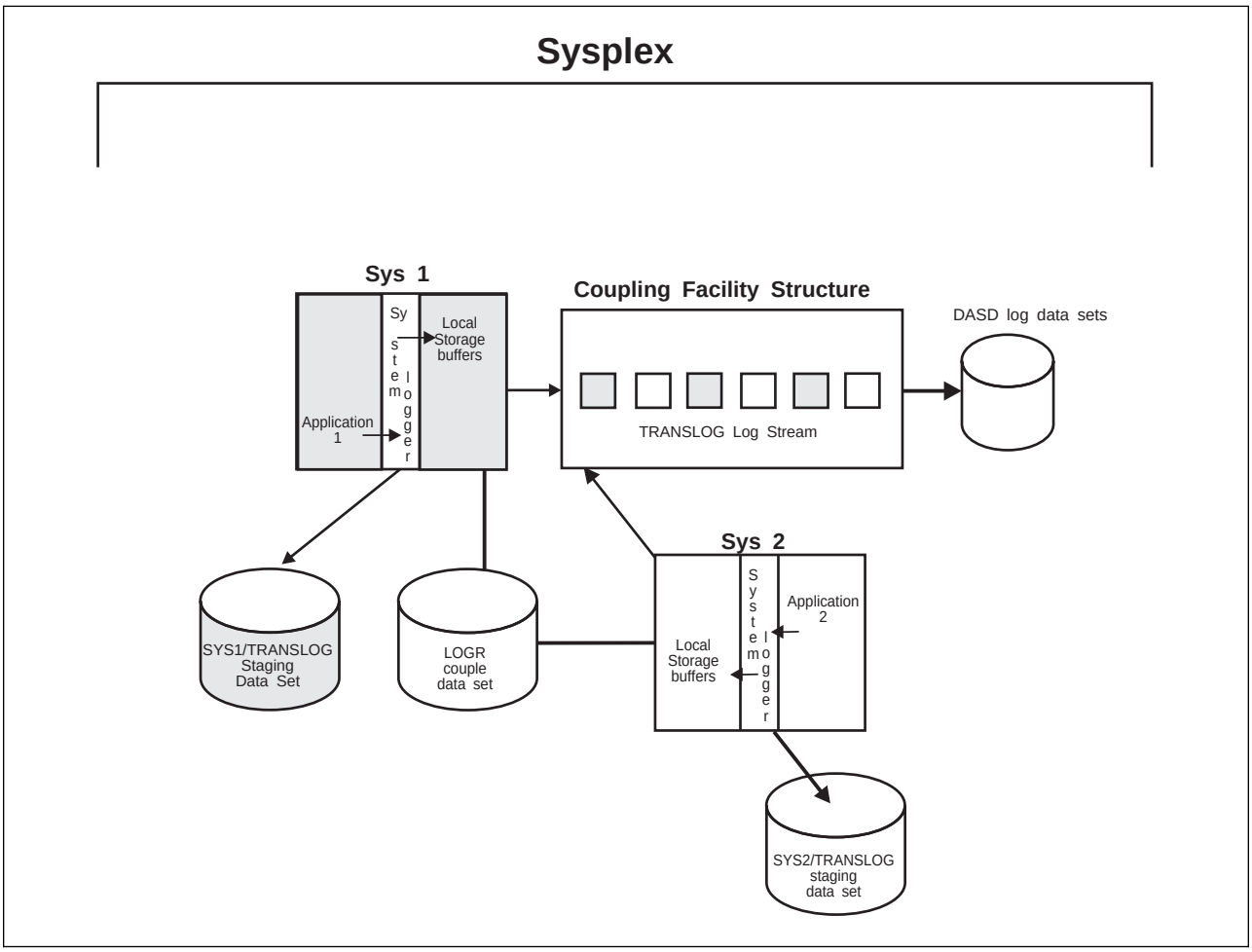


Figure 26-4. A Complete Coupling Facility Log Stream Configuration

DASD-Only Log Stream Configuration: Figure 26-5 on page 26-6 shows all the parts involved when a system logger application writes to a DASD-only log stream. System logger writes the data to the local storage buffers on the system, duplexing it at the same time to the DASD staging data sets associated with the log stream. Then, when the staging data set fills with data, system logger offloads the data to DASD log data sets. Note that where duplexing to DASD staging data sets is an option for a coupling facility log stream, it is a required automatic part of a DASD-only log stream. A system logger application uses system logger services to access the system logger capabilities.

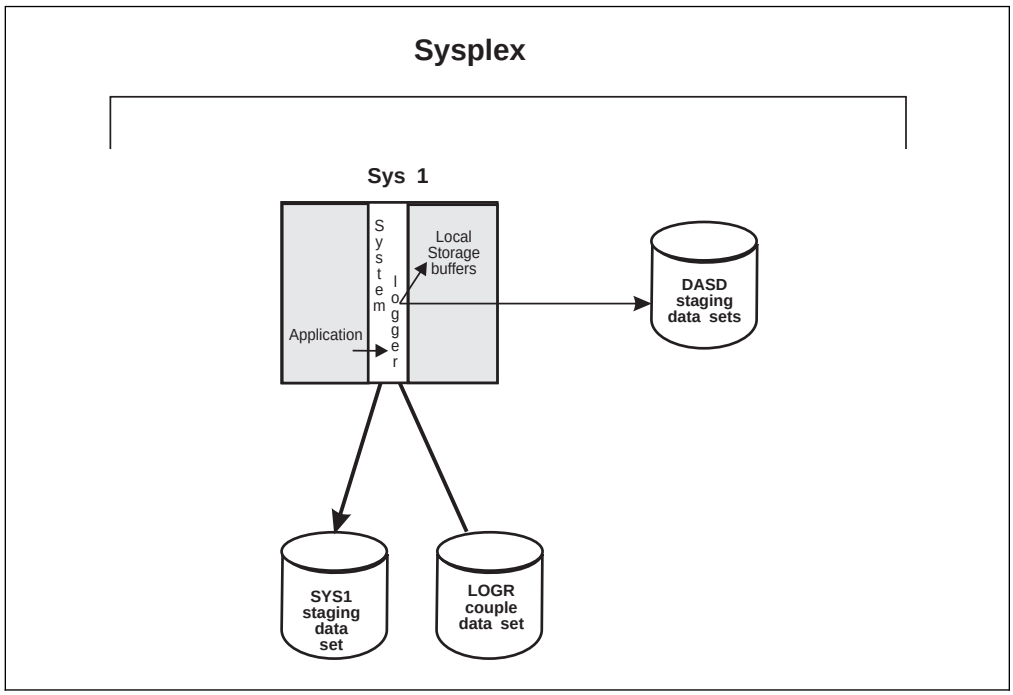


Figure 26-5. A DASD-Only Configuration

The System Logger Component

The system logger component resides in its own address space on each system in a sysplex. Some of the component processing will differ, depending on whether a given log stream is a coupling facility log stream or a DASD-only log stream. The system logger component does the following:

- Provides a set of system services that allows a system logger application to use the system logger component. See *OS/390 MVS Programming: Assembler Services Guide* and *OS/390 MVS Programming: Assembler Services Reference*.
- Maintains information in the LOGR policy about the current use of log streams and if used, coupling facility list structures.
- For coupling facility log streams, system logger interacts with cross-system extended services (XES) to connect to and use the coupling facility for system logger applications.
- Obtains local storage buffer space. For a coupling facility log stream, local storage buffers can be used for duplexing log data. For a DASD-only log stream, local storage buffers are used as interim storage for log data before it is offloaded to DASD log data sets.
- Offloads data to DASD log data sets as follows:

  For coupling facility log streams, system logger offloads log data from the coupling facility to DASD log data sets as the coupling facility structure space associated with the log stream reaches the installation-defined thresholds.

  For DASD-only log streams, system logger offloads log data from the local storage buffers to DASD log data sets as the DASD staging data set space reaches the installation-defined thresholds.
- Automatically allocates new DASD log data sets for log streams.

- Maintains a backup copy of (duplexes) log data that is in interim storage for recovery. Log data in interim storage is vulnerable to loss due to system or sysplex failure because it has not yet been hardened to DASD log data sets. System logger duplexes interim storage log data for both coupling facility and DASD-only log streams.
- Produces SMF record type 88 for system logger accounting on a single system. Record type 88 focuses on the usage of interim storage (coupling facility or local storage buffers) and log stream data for a system in the sysplex. Using the record can help an installation avoid the STRUCTURE FULL or STAGING DATA SET FULL exceptions, and perform other tuning and/or capacity planning analysis.

  See *OS/390 MVS System Management Facilities (SMF)* for more information on record type 88 and system logger accounting. Sample program IXGRPT1 in SYS1.SAMPLIB shows an example of producing a report from SMF record type 88.
- Ensures that:
  - When the last connection from a system disconnects from the log stream, all log data written by that system to the log stream is offloaded to DASD log data sets.

    System logger also deletes any staging data sets in use for a system at this time.
  - When the last connection to a coupling facility log stream in the sysplex disconnects, all coupling facility log data is offloaded to DASD log data sets and the coupling facility space is returned to XES for reallocation.
- Provides recovery support in the event of application, system, sysplex, or coupling facility structure failure for coupling facility log streams. (See "Recovery Performed for DASD-Only Log Streams" on page 26-54 for information about recovery for DASD-only log streams.)

The LOGR Couple Data Set (LOGR Policy)

The system logger component manages log streams based on the policy information that installations place in the LOGR couple data set. The LOGR couple data set must:

- Be accessible by all the systems in the sysplex.
- Be formatted using the IXCL1DSU utility.
- Contain policy information, which is defined using the IXCMIAPU utility (or the IXGINVNT service, see *OS/390 MVS Programming: Assembler Services Guide* for information).

You can only have one LOGR policy for the sysplex, even if you have primary and alternate LOGR couple data sets. System logger will only use the policy information contained in the primary LOGR couple data set. Because there is only one LOGR policy allowed for a sysplex, you cannot specify the DEFINE POLICY NAME parameter in the LOGR policy, nor can you issue the SETXCF START,POLICY command to activate a policy. Updates made to the LOGR policy are made in the primary LOGR couple data set. If you switch to your alternate LOGR couple data set, system logger copies the LOGR policy from the old primary couple data set into the new one.

The LOGR policy includes the following:

- Log stream definitions.
- Coupling facility list structure definitions, if applicable.
- Data containing the current state of a log stream (for example, whether a log stream is currently connected to the coupling facility structure).

Formatting a LOGR couple data set and defining the LOGR policy are steps an installation must take to prepare for using a system logger application. See “Setting Up the System Logger Configuration” on page 26-47.

Log Data on The Coupling Facility

For a coupling facility log stream, system logger requires that a coupling facility list structure be assigned for each log stream as interim storage. When an application writes data to a coupling facility log stream, it is first written to the coupling facility list structure for that log stream. System logger manages connections to the coupling facility, but the installation must define coupling facility structures dedicated to log streams in the LOGR policy as part of the planning process for system logger applications. More than one log stream can share space in a single structure.

System logger will dynamically manage the entry-to-element ratio for log stream coupling facility structures if you are using a LOGR primary couple data set formatted at the OS/390 Release 3 level or above. The installation defines an initial average buffer size for a structure on the AVGBUFSIZE parameter in the LOGR policy using the IXGINVNT service or the IXCMIAPU utility. System logger uses this value to determine the initial entry-to-element ratio for the structure. After that, system logger will dynamically manage the entry-to-element ratio for the structure based on actual structure usage. The most current entry-to-element ratio is used on any subsequent allocations of the structure.

Log Data on DASD Log Data Sets

System logger uses VSAM linear data sets to store log stream data that has been moved from the coupling facility when the coupling facility structure space allocated for the log stream reaches its installation defined threshold. See *OS/390 MVS Setting Up a Sysplex* for more information on this topic.

A log stream can have data in multiple DASD log data sets; as a log stream fills log data sets on DASD, system logger automatically allocates new ones for the log stream. System logger increments the sequence number in the log stream data set name as new data sets are added for a particular log stream. See the system logger chapter in *OS/390 MVS Setting Up a Sysplex* for information on planning naming conventions for system logger resources.

IBM recommends that DASD log data sets be managed by System Managed Storage (SMS). You can manage log stream data sets by either:

- Modifying automatic class selection (ACS) routines.
- Defining the SMS data class, storage class, and management classes explicitly in the log stream definition using the IXCMIAPU utility or the IXGINVNT service.

System logger will tolerate use of DFHSM or another product to migrate DASD log data sets.

Duplexing Log Data

Every time a system logger application writes a log block to a log stream, system logger automatically makes a duplicate copy of the data as insurance against data loss due to coupling facility, system failure, or system logger failure. System logger keeps a duplex copy of data in log stream interim storage only. The duplicate copy is kept until the data is offloaded from interim storage to DASD log data sets.

The way system logger duplexes log data depends on whether the log stream is a coupling facility or a DASD-only log stream.

Duplexing Log Data on a Coupling Facility Log Stream

System logger allows the installation to choose between two methods of duplexing coupling facility log data for a coupling facility log stream:

- Maintain a copy of coupling facility resident log data in local storage buffers on each system.

  This option is the default, and occurs automatically if you do not specifically request duplexing on staging data sets in the log stream definition in the LOGR policy.

- Maintain a copy of coupling facility resident log data in DASD **staging data sets**, one per log stream on a system.

  This option provides hardening of the data on failure independent, persistent media.

You can request the duplexing options for a coupling facility log stream in the log stream definition in the LOGR policy.

You can request duplexing to staging data sets either conditionally, so that a connection will have a staging data set when it contains a single point of failure, or unconditionally, so that every connection to a log stream duplexes to staging data sets. This means that the number of staging data sets active at any given time will depend on the status of each connection and the duplexing method selected for each log stream.

If you request duplexing using staging data sets for a coupling facility log stream in the LOGR policy, system logger uses VSAM linear DASD staging data sets to hold a back-up copy of coupling facility log data for log streams that specify it.

IBM recommends that you plan for DASD staging data sets even if you do not request duplexing to staging data sets. If the primary LOGR couple data set for the sysplex is at Release 3 level or above, system logger will automatically allocate a staging data set for a log stream under specific error conditions where the log data exists only in a volatile configuration. System logger does this to help safeguard log data against data loss.

Duplexing for a DASD-Only Log Stream

For a DASD-only log stream, system logger automatically duplexes log data from the system's local storage buffers to DASD staging data sets. System logger uses VSAM linear DASD data sets for staging data sets. Note that the staging data set naming convention is different for DASD-only and coupling facility log streams.

Overview of System Logger Services

This section provides an overview of general information about system logger services, including:

- "Summary of System Logger Services."
- "Define Authorization to System Logger Resources" on page 26-11.
- "Synchronous and Asynchronous Processing" on page 26-11.
- "Coding a System Logger Complete Exit for IXGBRWSE, IXGWRITE, and IXGDELET" on page 26-13.
- "How System Logger Handles Gaps in the Log Stream" on page 26-15.
- "Using the System Logger Answer Area (ANSAREA parameter)" on page 26-18.
- "Using ENF Event Code 48 in System Logger Applications" on page 26-19.

Summary of System Logger Services

System logger provides the following set of services:

IXGINVNT Define and maintain log stream and coupling facility structure information in the LOGR policy dynamically. See page 26-20.

You can also use the IXCMIAPU utility to specify log stream and structure definitions in the LOGR policy. IXCMIAPU also enables you to request a report of current log stream definitions.

IXGCONN Connect and disconnect an application to and from a log stream. See page 26-22.

IXGWRITE Write user-defined log data to a log stream. See page 26-32.

IXGBRWSE Browse (read) data from a log stream. See page 26-34.

IXGDELET Delete data from a log stream. See page 26-38.

IXGIMPRT Import (write) log blocks to a log stream, with a log block identifier and timestamp. See page 26-39.

IXGQUERY Retrieve information from a log stream. See page 26-41.

IXGOFFLD Initiate an offload of log data from the coupling facility structure for coupling facility log streams and from local storage buffers for DASD-only log streams to DASD log data sets. See page 26-45.

IXGUPDAT Modify the GMT time stamp maintained in the control information for a log stream. See page 26-46.

The following services contain parameters for both authorized and unauthorized programs: IXGCONN, IXGBRWSE, IXGWRITE, and IXGDELET. All other system logger services and their parameters can be used by any program. All the guidance for system logger, authorized and unauthorized, is contained in this chapter.

Define Authorization to System Logger Resources

Installations need to define authorization to system logger resources for both the system logger address space and logging functions and applications using the system logger services:

- For information on authorization for the system logger address space, see the chapter on planning for system logger functions in *OS/390 MVS Setting Up a Sysplex*.
- If you are using the IXCMIAPU utility to update the LOGR couple data set, you must define authorization to system logger resources for IXCMIAPU users. See the chapter on planning for system logger functions in *OS/390 MVS Setting Up a Sysplex*.
- To define authorization for system logger application programs, see "Authorization for System Logger Application Programs."

Authorization for System Logger Application Programs

IBM recommends that installations use Security Authorization Facility (SAF) to control access to system logger resources such as log streams or coupling facility structures associated with log streams.

Define access for applications to the following classes and resources for each service. Note that only applications writing to coupling facility log streams need access to coupling facility structures:

Figure 26-6. Defining SAF Authorization For System Logger Resources

System Logger Service	Access type	SAF class and Resource
IXGINVNT REQUEST=DEFINE TYPE=LOGSTREAM IXGINVNT REQUEST=UPDATE TYPE=LOGSTREAM IXGINVNT REQUEST=DELETE TYPE=LOGSTREAM	ALTER	RESOURCE(*log_stream_name*) CLASS(LOGSTRM)
IXGINVNT REQUEST=DEFINE TYPE=LOGSTREAM STRUCTNAME=*structname*	ALTER	RESOURCE(*log_stream_name*) CLASS(LOGSTRM)
	UPDATE	RESOURCE(IXLSTR.*structure_name*) CLASS(FACILITY)
IXGINVNT REQUEST=DEFINE TYPE=LOGSTREAM LIKE=*like_log_stream_name*	ALTER	RESOURCE(*log_stream_name*) CLASS(LOGSTRM)
	UPDATE	RESOURCE(*like_log_stream*) CLASS(LOGSTRM) RESOURCE(IXLSTR.*like_structure_name*) CLASS(FACILITY)
IXGINVNT REQUEST=DEFINE TYPE=STRUCTURE IXGINVNT REQUEST=DELETE TYPE=STRUCTURE	ALTER	RESOURCE(MVSADMIN.LOGR) CLASS(FACILITY)
IXGCONN REQUEST=CONNECT AUTH=WRITE	UPDATE	RESOURCE(*log_stream_name*) CLASS(LOGSTRM)
IXGCONN REQUEST=CONNECT AUTH=READ	READ	RESOURCE(*log_stream_name*) CLASS(LOGSTRM)

Synchronous and Asynchronous Processing

Depending on the operating environment, an event control block (ECB) can be resident in one of two places: the home address space (in private storage) or in common storage. The following section provides examples of where the ECB might appear in an environment that includes a caller address space, a server address space, and the system logger address space. In these examples, the server provides services to the caller.

Use the MODE parameter on the IXGWRITE, IXGBRWSE, and IXGDELET services to choose one of the following:

- MODE=SYNC
- MODE=SYNCECB
- MODE=SYNCEXIT
- MODE=ASYNCNORESPONSE

- **Choosing MODE=SYNC**

  Choose MODE=SYNC to specify that the request be processed synchronously. When control returns to the caller, all processing on behalf of the request is complete.

- **Choosing MODE=SYNCECB**

  Choose MODE=SYNCECB to specify that the request be processed synchronously, **if possible**. If the system logger request cannot be completed synchronously, processing on behalf of the request might still be in progress when control returns to the caller. When processing for the request completes, the ECB specified on the system logger request is posted. Once the ECB is posted, the caller can examine the answer area to verify whether the request completed successfully.

- **Choosing MODE=SYNCEXIT**

  Choose MODE=SYNCEXIT to specify that the request be processed synchronously, **if possible**. If the request cannot be completed synchronously, control returns to the caller with a return and reason code indicating that processing of the request is not complete. When processing of the request completes, the exit routine specified at connect time gets control. (The exit routine is specified on the COMPLETEEXIT parameter on the IXGCONN request.)

  You can use the REQDATA parameter with MODE=SYNCEXIT to specify user-defined information relating to the request. For example, you can use REQDATA to specify information about control blocks needed by the complete exit.

  Your application must be in supervisor state, system key to use the MODE=SYNCEXIT and REQDATA parameters.

- **Choosing MODE=ASYNCNORESPONSE**

  Choose MODE=ASYNCNORESPONSE on the IXGWRITE and IXGDELET requests to specify that the request be processed asynchronously. The caller will not be informed when the request completes. The answer area returned in the ANSAREA parameter and mapped by IXGANSAA is not valid when you specify MODE=ASYNCNORESPONSE.

When a system logger request cannot be completed synchronously, system logger schedules an SRB to complete processing of the request before it returns control to the caller. While the SRB runs independent of the requesting task, the SRB might encounter an error from which it cannot recover. The SRB ensures that the error condition is percolated to the task that issued the system logger request.

Note: Depending on the exploiter's structure, this task might not be the same task that originally issued the IXGCONN request to connect to a log stream.

Prior to percolating the error to the requesting task, system logger issues the SETRP macro, specifying SERIAL=YES. System logger also places additional diagnostic information in the SDWA, as follows:

SDWACMPC
: The completion code, set to X'1C5'.

SDWACRC
: The reason code, set to X'85F'.

SDWACOMU
: The ECB address specified on the ECB keyword when system logger was invoked.

If the caller receives a return code indicating that system logger will process the request asynchronously, the application cannot free certain storage areas.

Reference: See *OS/390 MVS Programming: Assembler Services Reference* for the specific service's return code.

Coding a System Logger Complete Exit for IXGBRWSE, IXGWRITE, and IXGDELET

Your complete exit provides a mechanism for system logger services to let you know when your asynchronously processed system logger IXGBRWSE, IXGDELET, or IXGWRITE request completes. You provide the address of your complete exit using the COMPLETEEXIT parameter when you issue the IXGCONN REQUEST=CONNECT macro to connect to the log stream. You will be informed of request completion through your complete exit if you specify MODE=SYNCEXIT and the system processes your request asynchronously.

You must be running in supervisor state and a system key to specify MODE=SYNCEXIT on a IXGBRWSE, IXGDELET, or IXGWRITE system logger service request.

Information Passed to the Complete Exit.

When the complete exit gains control, it receives the following information about the system logger request in the complete exit parameter list (CMPL), mapped by the IXGCMPL macro:

CMPLREQDATA
: Information passed to the complete exit by the issuer of the system logger service request. The use of this optional field is user-defined.

CMPLRETCODE
: Return code from the system logger service request.

CMPLRSNCODE
: Reason code from the system logger service request.

CMPLANSAREA@
: Answer area address. The answer area is mapped by IXGANSAA macro.

CMPLSTREAMTOKEN
: Log stream connector's stream token.

Environment

The complete exit receives control in the following environment:

Authorization:	Supervisor state, and PSW key 0
Dispatchable unit mode:	SRB
Cross memory mode:	PASN=HASN=SASN. PASN, HASN and SASN are equal to the PASN at the time of the connect to the log stream.
AMODE:	31-bit
ASC mode:	Primary ASC mode
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held.
Control parameters:	None.

Input Specifications

System logger services pass information to the complete exit in registers.

Registers at Entry: When the complete exit receives control, the GPRs contain the following information:

Register	Contents
0	Does not contain any information for use by the complete exit.
1	Address of a fullword containing the address of the CMPL
2-12	Do not contain any information for use by the complete exit.
13	Address of a standard 72-byte save area.
14	Return address to system logger services.
15	Entry point address.

When the complete exit receives control, the ARs contain no information for use by the complete exit.

Return specification: Your exit must return control to the system by branching to the address provided on entry in register 14. There are no requirements for the GPRs or ARs to contain any particular value.

Programming Considerations

If you have more than one outstanding system logger request being processed asynchronously, multiple instances of your complete exit might run concurrently as system logger services process your request completions. Therefore, you should consider coding your complete exit as a reentrant program.

You can access the input parameter data area only while your complete exit is running. If you want to save the parameter information for later processing, make a copy of it before your complete exit returns control to the system.

In certain instances, the system must quiesce the activity of user exits in order to perform cleanup processing. The following illustrates scenarios where this processing occurs:

- Connection Termination

When a user disconnects while a MODE=SYNCEXIT request is outstanding, the complete exit will not be called.

If the connecting task terminates, the system will issue a PURGEDQ against SRB that are associated with the connection. Since all complete exit SRBs are associated with the connecting task's TCB, any complete exits that are active when the connecting task terminates could be interrupted with an ABEND x'47B' reason code 0.

- System Logger Termination

  When the system logger address space terminates, it attempts to inform active requests of logger termination. If the complete exit has not been scheduled, it is scheduled at this time with a return code and reason code indicating that the system logger address space has terminated.

The system logger initializes a recovery environment before it calls the complete exit. Should the complete exit fail and percolate to the system logger's recovery routine, the task that did the corresponding connection to the log stream is abended and retry is not permitted. The abend code will be 1C5 and the abend reason code 00030006.

How System Logger Handles Gaps in the Log Stream

System logger might find data unexpectedly missing from or inaccessible in a log stream. These areas of missing information are called gaps and result from the following:

- System, sysplex, or coupling facility failure where the data could not be recovered.
- A DASD log stream data set being deleted.
- DASD I/O errors occurring during processing of an IXGBRWSE request to read log data.

If system logger encounters a gap during the processing of an IXGBRWSE or IXGDELET service, it returns a return and reason code to the caller of the system logger service to report the condition and indicate whether the service completed successfully. The other system logger services, IXGINVNT, IXGCONN, and IXGWRITE, are not affected by gaps in the log stream data.

See Figure 26-7 and Figure 26-8 on page 26-16 for a summaries of how IXGBRWSE and IXGDELET services handle gaps in the log stream.

Figure 26-7 (Page 1 of 2). How IXGBRWSE Requests Handle Gaps in a Log Stream

IXGBRWSE Request	**Request Results**
START SEARCH=*search* READBLOCK SEARCH=*search*	Request completes with a non-zero return code and reason code. System logger positions the cursor at or reads the next youngest log block or a block with an identical time stamp, if there is one.
START STARTBLOCKID=*startblockid*	Request completes with a non-zero return code and reason code. System logger positions the cursor at or reads the next youngest log block, if one exists.
READBLOCK BLOCKID=*blockid*	Request fails with a non-zero return and reason code. No block is returned.

Figure 26-7 (Page 2 of 2). How IXGBRWSE Requests Handle Gaps in a Log Stream

IXGBRWSE Request	Request Results
READCURSOR	Request completes with a non-zero return code and reason code. System logger reads the next valid block after the gap in the specified direction, if there is one.
START OLDEST RESET POSITION=OLDEST	When the oldest log block is within a gap of log data, the request completes with a non-zero return code and reason code. System logger positions or resets the cursor to the oldest valid block in the log stream.
START YOUNGEST RESET POSITION=YOUNGEST	The service completes without errors.

Figure 26-8. How IXGDELET Requests Handle Gaps in a Log Stream

IXGDELET Request	Request Results
BLOCKS=ALL	This request is unaffected by gaps in the log stream. The service completes successfully.
BLOCKS=RANGE	If the block specified is at the start of the range specified, the service fails with a non-zero return and reason code. No data is deleted. System logger returns the block identifier of the first accessible block toward the young end of the log stream. If the block specified is **not** within a gap, the service completes successfully.

Dumping on Data Loss (804–type) Conditions

The new DIAG option on the IXGINVNT service and IXCMIAPU TYPE(LOGR) utility along with the new DIAG options on the IXGCONN, IXGDELET and IXGBRWSE services allow additional diagnostic data to be obtained when a log stream loss of data condition is encountered.

The following shows the default settings for all the services and the relationship between the service specifications in terms of whether or not Logger will request a dump for certain loss of data type conditions.

Default settings:

```
DEFINE LOG STREAM     DIAG(NO)
 IXGCONN (CONNECT)     DIAG(NO_DIAG)
  IXGBRWSE (START)      DIAG(NO_DIAG)
  IXGDELET              DIAG(NO_DIAG)
```

Assuming a data loss (804) type condition was encountered on an IXGDELET request or on any IXGBRWSE request for a browse session, the following shows whether or not a dump would be taken by Logger for this condition, where:

No No dump is taken for the condition.

Yes Dump is taken for the condition.

DEFINE LOG STREAM		NO			YES		
IXGCONN CONNECT		NO_DIAG	NO	YES	NO_DIAG	NO	YES
IXGBRWSE START	NO_DIAG	no	no	no	no	no	yes
	NO	no	no	no	no	no	no
	YES	no	no	no	yes	no	yes
IXGDELET	NO_DIAG	no	no	no	no	no	yes
	NO	no	no	no	no	no	no
	YES	no	no	no	yes	no	yes

Define a Log Stream to Allow Additional Dumping

- IXGINVNT

  Specify IXGINVNT REQUEST=DEFINE,TYPE=LOGSTREAM,...,DIAG=YES

- IXCMIAPU Utility Program

  Specify DEFINE LOGSTREAM ... DIAG(YES)

Define a Log Stream to Allow Additional Dumping Using LIKE

- IXGINVNT

  Specify IXGINVNT
  REQUEST=DEFINE,TYPE=LOGSTREAM,NAME=like.log.stream,...,DIAG=YES

  Then IXGINVNT
  REQUEST=DEFINE,TYPE=LOGSTREAM,LIKE=like.log.stream

- IXCMIAPU Utility Program

  Specify DEFINE LOGSTREAM NAME (like.log.stream) ... DIAG(YES)

  Then DEFINE LOGSTREAM LIKE(like.log.stream)

Update a Log Stream to Allow Additional Dumping

- IXGINVNT

  Specify IXGINVNT REQUEST=UPDATE,TYPE=LOGSTREAM,...,DIAG=YES

- IXCMIAPU Utility Program

  Specify UPDATE LOGSTREAM ... DIAG(YES)

The update will take affect as each system obtains its first connection to the log stream. For example, assume there are two systems in an installation's sysplex, SYSA and SYSB, and SYSA had a connection to the log stream prior to the update request and SYSB did not have any connections to the log stream. If SYSB establishes a new (first) connection to the log stream, then the DIAG options will be in affect on SYSB. The DIAG settings prior to the update request will remain in affect on SYSA even if there is another connection to the log stream on SYSA. However, if all the connections to the log stream on SYSA are disconnected and then a new (first) connection is established to the logstream on SYSA, then the new DIAG options would then be in affect on SYSA.

Connect to a Log Stream and Request Additional Dumping

- IXGCONN

  Specify IXGCONN REQUEST=CONNECT,...,DIAG=YES

Assuming the log stream that is being connected has been defined as DIAG(YES), then Logger will provide a dump when a loss of data condition is encountered on browse and delete requests, unless the specific browse (start) or delete request specifies otherwise.

Browsing a Log Stream and Request Additional Dumping

- IXGBRWSE

  Specify IXGBRWSE REQUEST=START,...,DIAG=YES when IXGCONN REQUEST=CONNECT,...,DIAG=NO_DIAG

- IXGCONN

  Specify IXGCONN REQUEST=CONNECT,...,DIAG=YES then no specific DIAG options need to be coded on the IXGBRWSE REQUEST=START invocation.

Assuming the log stream that is being browsed has been defined as DIAG(YES), then Logger will provide a dump when a loss of data condition is encountered on any browse for this browse session.

Deleting Log Data From a Log Stream and Request Additional Dumping

- IXGDELET

  Specify IXGDELET,...,DIAG=YES when IXGCONN REQUEST=CONNECT,...,DIAG=NO_DIAG

- IXGCONN

  Specify IXGCONN REQUEST=CONNECT,...,DIAG=YES then no specific DIAG options need to be coded on the IXGDELET invocation.

Assuming the log stream that is having log data deleted has been defined as DIAG(YES), then Logger will provide a dump when a loss of data condition is encountered on this delete request.

Using the System Logger Answer Area (ANSAREA parameter)

Every system logger service issued must include an answer area output field specified on the ANSAREA parameter. In this answer area, mapped by the IXGANSAA macro, system logger returns status and diagnostic data.

Some answer area fields include:

ANSAA_PREFERRED_SIZE
The optimal size for the answer area field specified on the ANSAREA parameter. IXGCONN REQUEST=CONNECT returns this value.

The answer area must be at least 40 bytes long. If you specify a field that is less than that, the service fails with a non-zero return code. To ascertain the optimal answer area size, look at the ANSAA_PREFERRED_SIZE field of the answer area returned by the first system logger service your application issues, either the IXGINVNT or IXGCONN REQUEST=CONNECT service.

ANSAA_DYNMGMTOFENTRYTOELACTIVE
Indicates that system logger is dynamically managing the entry-to-element ratio for the coupling facility structure. The structure was defined in a LOGR couple data set at an OS/390 Release 3 level or higher.

ANSAA_ASYNCH_RETCODE
For asynchronously processed requests, system logger returns the return code in this field.

ANSAA_ASYNCH_RSNCODE
For asynchronously processed requests, system logger returns the reason code in this field.

ANSAA_DIAGx
Diagnostic data. The content of these fields depend on any return and reason codes returned for a request.

ANSAA_DASDONLYLOGSTREAM
This flag in ANSAA_FLAGS1 indicates whether a log stream is DASD-only. It is output from an IXGCONN REQUEST=CONNECT service.

See *OS/390 MVS Data Areas, Vol 3 (IVT-RCWK)* for a complete description of the IXGANSAA macro.

Using ENF Event Code 48 in System Logger Applications

System logger issues ENF event code 48 to broadcast status changes in the system logger address space, log streams, and coupling facility structures. Since these status changes can affect the outcome of system logger service requests, **IBM recommends** that you use ENF event code 48 to receive notification of these status changes, using the ENFREQ service to listen for event 48. Note that your program must be authorized to use the ENFREQ service. Applications should issue the ENFREQ service to listen for event 48 before connecting to a log stream.

For example, suppose an IXGWRITE request fails with a return and reason code indicating that some system logger resource is unavailable, perhaps because the system logger address space has failed or because a structure rebuild is in progress for the coupling facility structure for the log stream. Before the application can resume issuing system logger service requests. it must listen for the event code 48 notification that the resource is available again.

In order to listen for ENF code 48 events, you must code an SRB-type listen exit for event code 48 events to scan the event 48 parameter list for status information on the system logger component, log streams, and coupling facility structures associated with log streams. The listen exit must be in place before system logger applications are activated.

If you use ENF event code 48 to receive information about system logger events, make sure that you take into account the asynchronous nature of the ENF exit. You might get notified of events out of sequence, being notified for instance, that a problem has been resolved before you get a return and reason code describing a problem.

For example, if you issue IXGWRITE to write data to the log stream while the coupling facility structure space allocated for the log stream is full, you might get an

ENF 48 notification that the structure is no longer full before you get the return and reason code from IXGWRITE to say that the structure is full.

Applications that do not want to use ENF event code 48 or that are unauthorized and cannot use ENFREQ will still receive logger service return and reason codes indicating failure or resource shortages. These applications can then simply set a timer and then retry the requests to see if the problem has resolved itself.

References:

- See "Writing an ENF Event 48 Listen Exit" on page 26-47 for information on ENF 48 events, and coding your ENF event 48 listen exit.
- See *OS/390 MVS Programming: Assembler Services Guide* for guidance about using the ENFREQ macro.
- See *OS/390 MVS Programming: Authorized Assembler Services Reference ENF-IXG* for reference information on the ENFREQ macro.

Figure 26-9 shows an example of how to issue the ENFREQ service to listen for ENF event code 48 and specify a listen exit to analyze the event 48 parameter list:

```
ENFREQ ACTION=LISTEN,
       CODE=ENFPC048,
       ESTBNME=THISMOD,
       EXITNME=LOGLISTEN,
       SRBEXIT=(R02),
       EOM=YES,
       DTOKEN=ENFREQ_DTOKEN,
       RETCODE=ENFREQ_RETCODE
```

Figure 26-9. Issuing ENFREQ to Listen for ENF Event Code 48

IXGINVNT: Managing the LOGR Policy

The IXGINVNT service allows your application program to manage and update the LOGR policy dynamically. You can also use the IXCMIAPU utility to manage the LOGR policy (see *OS/390 MVS Setting Up a Sysplex*). The reason and return codes for both the IXGINVNT service and the IXCMIAPU utility are documented with the reference information on the IXGINVNT service in *OS/390 MVS Programming: Assembler Services Reference*.

For guidance on specifying information in the LOGR couple data set using IXGINVNT or IXCMIAPU, see the system logger chapter of *OS/390 MVS Setting Up a Sysplex*.

Using IXGINVNT, your application program can:

- Define and update log streams definitions using REQUEST=DEFINE or REQUEST=UPDATE with TYPE=LOGSTREAM.
- Define a coupling facility structure associated with a log stream using REQUEST=DEFINE with TYPE=STRUCTURE.
- Delete log stream and coupling facility structure definitions using REQUEST=DELETE.

Defining a Model Log Stream in the LOGR Couple Data Set

Use the MODEL parameter on the IXGINVNT REQUEST=DEFINE service to define a dummy log stream that other log stream definitions can reference. Using a model allows you to specify common log stream characteristics for many log streams only once.

This can streamline the process of defining new log streams in the LOGR policy, because you can specify the name of the model log stream on the LIKE keyword when you issue the IXGINVNT service to define a new log stream. Note that any explicit definitions for the log stream you are defining override the definitions of the model log stream.

Figure 26-10 shows how you can use the IXGINVNT service to define a log stream as a model and then define a log stream modeled after it. The LS_DATACLAS parameter specifying the data class for the log data sets for log stream STREAM1 overrides the LS_DATACLAS specified in the IXGINVNT request for log stream MODEL1.

```
IXGINVNT REQUEST=DEFINE
      TYPE=LOGSTREAM,
      STREAMNAME=MODEL1,
      STRUCTNAME=STRUCT1,
      MODEL=YES,
      HLQ=SYSPLEX1,
      STG_DUPLEX=YES,
      DUPLEXMODE=COND,
      STG_DATACLAS=STGDATA,
      STG_MGMTCLAS=STGMGMT,
      STG_STORCLAS=STGSTOR,
      LS_MGMTCLAS=LSMGMT,
      LS-STORCLAS=LSSTOR,
      LS_DATACLAS=LSDATA,
      STG_MGMTCLAS=LSMGMT,
      STG_STORCLAS=LSSTOR,
      ANSAREA=ANSAREA,
      ANSLEN=ANSLEN,
      RETCODE=RETCODE,
      RSNCODE=RSNCODE

IXGINVNT REQUEST=DEFINE
      TYPE=LOGSTREAM,
      STREAMNAME=STREAM1,
      LS_DATACLAS=LSDATA1
      LIKE=MODEL1,
      ANSAREA=ANSARE1,
      ANSLEN=ANSLE1,
      RETCODE=RETCDE,
      RSNCODE=RSNCDE
```

Figure 26-10. Define a Log Stream as a Model and then Model a Log Stream After It

You **cannot** connect to a model log stream or use it to store log data. It is strictly for use as a model to streamline the process of defining log streams.

You can use the LIKE parameter to reference a log stream that has not been defined as a model log stream using the MODEL parameter. Any log stream can be referenced on the LIKE parameter.

Defining a Log Stream as DASD-Only

You can define a log stream as either a coupling facility log stream or DASD-only log stream using the DASDONLY parameter on the IXGINVNT REQUEST=DEFINE TYPE=LOGSTREAM service (see “The Log Stream” on page 26-2).

Specify or default to DASDONLY=NO, to define a coupling facility log stream. You must also specify a structure name on the STRUCTNAME parameter in the log stream definition for a coupling facility log stream.

Specify DASDONLY=YES, to define a DASD-only log stream, which is not associated with a coupling facility structure. When you define a DASD-only log stream, you can also specify MAXBUFSIZE to define the maximum buffer size that can be written to the DASD-only log stream. (For a coupling facility log stream, MAXBUFSIZE is specified on the IXGINVNT REQUEST=DEFINE TYPE=STRUCTURE request to define the maximum log block size that can be written to the log stream.)

Note that a DASD-only log stream is single-system in scope - only one system may write to a DASD-only log stream.

For set up steps for a DASD-only log stream, see the system logger chapter in *OS/390 MVS Setting Up a Sysplex*.

Upgrading a Log Stream From DASD-Only to Coupling Facility

You can upgrade a log stream defined as DASD-only (DASDONLY=YES parameter on the log stream definition) to a coupling facility log stream. Do this by issuing the IXGINVNT REQUEST=UPDATE service with the STRUCTNAME parameter to associate a structure with the log stream.

To upgrade a log stream from DASD-only to coupling facility, the following must be true:

- The structure specified on the STRUCTNAME parameter must be defined in a structure definition in the LOGR policy.
- All connections (active or failed) to the log stream must be disconnected.

Before connecting to or issuing any system logger services against an upgraded log stream, you must also make sure that the structure associated with the log stream is specified in the CFRM policy couple data set.

For guidance on upgrading a DASD-only log stream to a coupling facility based one, see the system logger chapter in *OS/390 MVS Setting Up a Sysplex*.

IXGCONN: Connecting to and Disconnecting From a Log Stream

Use the IXGCONN service to connect to or disconnect from a log stream. An application must issue IXGCONN with REQUEST=CONNECT before it can read, write, or delete data in a log stream.

When the IXGCONN REQUEST=CONNECT request completes, it returns a unique connection identifier, called a **STREAMTOKEN**, to the calling program. The application uses the token in subsequent logger service requests to identify its connection to the log stream.

In the answer area (IXGANSAA) returned by IXGCONN, bit Ansaa_DynMgmtOffEntryToEleActive is on when system logger is dynamically managing the entry-to-element ratio. See the system logger chapter in *OS/390 MVS Setting Up a Sysplex* for information about dynamic management of the entry-to-element ratio.

Examples of Ways to Connect to the Log Stream

An application can connect to the log stream in different ways. Some of these are:

- **One Connection per Address Space**: Once a program has connected to a log stream, any task running in the same address space shares the connect status and can use the same stream token to issue other system logger services. Any task in the address space can disconnect the entire address space from the log stream by issuing the IXGCONN REQUEST=DISCONNECT service.
- **One Connection Per Program**: One or more tasks in a single address space can issue IXGCONN REQUEST=CONNECT individually to connect to the same log stream and receive separate stream tokens. Each program must disconnect from the log stream individually.
- **Multiple Systems Connecting to a Log Stream**: Multiple address spaces on one or more MVS systems can connect to a single coupling facility log stream, but each one **must** issue IXGCONN individually to connect and then disconnect from the log stream. Each one receives a unique stream token; address spaces cannot share a stream token.

When an application issues IXGCONN to connect to a coupling facility log stream, the system logger address space connects to the coupling facility list structure for the log stream.

Each task that issues IXGCONN REQUEST=CONNECT to connect to a log stream must later issue IXGCONN REQUEST=DISCONNECT to disconnect from the log stream. When a task disconnects from the log stream, the stream token that identified the connection is invalidated. Any requests that use the stream token after the disconnect will be rejected.

If a task ends before issuing a disconnect request, system logger will automatically disconnect the task from the log stream.

Additional Considerations for Connecting to a DASD-Only Log Stream

If you are writing an application for a DASD-only log stream, you must keep in mind that a DASD-only log stream is single-system in scope, meaning that only one system at a time can connect to a DASD-only log stream. (See “The Log Stream” on page 26-2 for a description of the two different types of log streams.) Multiple applications from the same system can connect to a DASD-only log stream, but only from one system at a time. An application trying to connect to a DASD-only log stream that already has another system connected will fail. A second system cannot connect to the DASD-only log stream until the first one disconnects.

The ANSAA_DASDONLYLOGSTREAM flag in the IXGANSAA mapping macro indicates whether a log stream is DASD-only.

When an application issues the IXGCONN request to connect to a DASD-only log stream, the STUCTNAME, AVGBUFSIZE, and ELEMENTSIZE will be returned

containing hexidecimal zeros, because they are all coupling facility structure related fields.

How System Logger Allocates Structure Space For a New Log Stream at Connection Time

The first IXGCONN request in a sysplex issued for a coupling facility log stream initiates the allocation of coupling facility space for the log stream.

If there are multiple coupling facility log streams assigned to one coupling facility structure, system logger divides the structure space evenly between each log stream that has applications connected to it via IXGCONN. This means that the first log stream that has an IXGCONN service issued against it gets 100% of the coupling facility structure space available. When a second log stream mapping to the coupling facility structure has an IXGCONN service issued against it, system logger divides the structure evenly between the two log streams. The first log stream's available coupling facility space will be reduced by half to reapportion the rest of the space to the second log stream. This can result in some of the data being off-loaded to DASD if the coupling facility space allocated to a log stream reaches its high threshold as a result of the reapportionment of the structure space.

For example, if an installation defines two log streams to a single structure in the LOGR policy, but only one of the log streams actually has a connected application, then that one log stream can use the entire coupling facility structure. When the second log stream connect occurs against the coupling facility structure, the space is gradually divided between the two log streams. Likewise, when the last disconnect is issued against one of the two log streams, all coupling facility structure space is again available to the first log stream.

Note that this process of reallocation of coupling facility space and subsequent offloading of coupling facility data might not take place immediately after the log stream connect. The reallocation of space might occur gradually, with an offloading taking place some time after the original log stream connect.

Connect Process and Staging Data Sets

Coupling Facility Log Streams: If an installation defines DASD staging data sets for a coupling facility log stream, the data sets are created, if necessary, during connection processing for the first IXGCONN request issued against a log stream from a particular system.

DASD-Only Log Stream: For a DASD-only log stream, the staging data set is automatically created during the first connect issued against the log stream. A staging data set is not optional for a DASD-only log stream.

Requesting Authorization to the Log Stream for an Application

Use the AUTH parameter to specify whether the application issuing IXGCONN should have read or write access to the log stream:

- If you specify AUTH=READ for an application, that application will only be able to issue IXGCONN, IXGBRWSE, and IXGQUERY requests. For AUTH=READ, the caller must have SAF read access to RESOURCE(*log_stream_name*) in CLASS(LOGSTRM).

- If you specify AUTH=WRITE for an application, that application can issue any system logger service. For AUTH=WRITE, the caller must have SAF UPDATE access to RESOURCE(*log_stream_name*) in CLASS(LOGSTRM).

Connecting as a Resource Manager

A resource manager is an application you can write and associate with a log stream to manage resources and processing for a log stream.

For example, via the resource manager user exit, a resource manager might be notified of a write or delete request issued against a log stream. The resource manager can then perform further processing to accept, reject, or override the delete request with a different log block identifier. A resource manager application can be helpful to perform any management functions on behalf of a log stream.

Before a resource manager can connect to a log stream, the name of the resource manager must be specified in the log stream definition in the LOGR couple data set. You can specify **one** resource manager name for a log stream in the log stream definition. See *OS/390 MVS Setting Up a Sysplex* for setting up a log stream definition in the LOGR couple data set.

If you specify a resource manager name for a log stream in the LOGR policy, the resource manager specified must connect to the log stream. If the resource manager does not connect, system logger will not process any IXGDELET requests to delete log data. This is so that the resource manager will not miss any information about deletes issued against the log stream.

The resource manager connects to the log stream it manages using the RMNAME, RMEXIT, RMDATA, and RMEVENTS parameters on the IXGCONN service. You must be running in supervisor state and a system key to use these parameters. The connect request must be issued from the resource manager address space. The resource manager address space must be non-swappable with an authorization index (AX) value of 1, or all invocations of the resource manager exit will fail.

Note that only one resource manager can connect to a log stream from a given system. The resource manager **can** connect to multiple log streams.

Use the resource manager parameters as follows:

RMNAME Specifies the name of the resource manager program connecting to the log stream. This is the same name specified on the RMNAME parameter in the LOGR couple data set log stream definition.

RMEXIT Specifies the name of a resource manager user exit. The resource manager exit is called when write and/or delete requests (as specified on the RMEVENTS parameter) are issued against the log stream that the resource manager manages. The RMEXIT keyword is required with RMNAME. For information on the resource manager exit, see “Coding a Resource Manager Exit for IXGCONN” on page 26-28.

RMEVENTS Specifies that write or delete requests issued against the log stream are to trigger the resource manager exit. RMEVENTS is required with RMNAME. You can specify RMEVENTS=LBWRITE, RMEVENTS=LBDELETE, or RMEVENTS=(LBWRITE,LBDELETE).

RMDATA Specifies user-defined data to the resource manager. This data is then passed to the resource manager user exit when the exit is called.

RMDATA is required with RMNAME.

Using ENF Event Code 48 With a Resource Manager

System logger issues many ENF event code 48 of use to a resource manager application.

See:

- "Using ENF Event Code 48 in System Logger Applications" on page 26-19 for using ENF event code 48 events.
- "Writing an ENF Event 48 Listen Exit" on page 26-47 for ENF event code 48 events and coding an ENF listen exit.

Requesting a Write or Import Connection - IMPORTCONNECT parameter

Use the IMPORTCONNECT parameter to specify whether a connection is a write or an import connection. The IMPORTCONNECT parameter is only valid with AUTH=WRITE.

- **A write connection**, (AUTH=WRITE IMPORTCONNECT=NO) specifies that you want to use the IXGWRITE request to write to a log stream. You must specify or default to IMPORTCONNECT=NO at connect time in order to use the IXGWRITE request against a log stream.

  You can have multiple write connects against a log stream, but only if there is no import connection established for that log stream anywhere in the sysplex. Once one or more applications connect to a log stream as a write connection, any subsequent attempts to connect as an import connection will fail.

  You cannot use the IXGIMPRT service against a log stream that has been connected to as a write connection from anywhere in the sysplex.

  An import connection, (AUTH=WRITE IMPORTCONNECT=YES) specifies that you want to use the IXGIMPRT request to import log data from one log stream to another, maintaining the same log block identifier and GMT time stamp. You might do this to create a copy of a log stream. You must specify IMPORTCONNECT=YES at connect time in order to use the IXGIMPRT request against a log stream.

  You can have only one import connection for a log stream from anywhere in the sysplex. An import connection cannot coexist with a write connection. Once an application connects to a log stream as an import connect, all subsequent AUTH=WRITE IMPORTCONNECT=YES|NO connect requests will fail.

  You cannot use the IXGWRITE service against a log stream that has been connected to as an import connection.

You cannot have both a write and an import connection to a log stream and you cannot issue both write and import requests to the same log stream.

Specifying User Data for a Log Stream

System logger allows 64 bytes of user specified data to be associated with a log stream and stored in the LOGR policy. The user data can be:

- Specified or changed using the USERDATA parameter on the disconnect request of IXGCONN.
- Read using the USERDATA output parameter on the connect request of IXGCONN.

If an application codes the USERDATA parameter on a connect request when there is no user data associated with the log stream, IXGCONN returns a USERDATA field containing all zeros.

Only one copy of the log stream user data exists and any application connected to the log stream can update this copy when they disconnect. If you will have multiple connectors to a log stream, you should consider using serialization or another protocol to protect the USERDATA field.

Using ENF Event 48 When a Connect Request is Rejected

If a connect request is rejected applications should listen for ENF 48 event informing listeners that the condition has changed or the problem has been resolved. Upon receipt of the ENF 48 event indicating that the problem is resolved, the application can retry the request.

System Logger Processing at Disconnection

Disconnection for an Application

When an application issues IXGCONN REQUEST=DISCONNECT to disconnect from a log stream, system logger rejects any new requests from that application.

If the application disconnects with outstanding asynchronous requests, the disconnect is accepted. Asynchronous requests then complete without notifying the disconnecting application.

Last Disconnection for Log Stream on a System

System logger rejects any new requests from the system for that log stream. All the log data from this system is then offloaded to DASD log stream data sets. This may include log data written from other systems connected to the log stream. For coupling facility log streams, the coupling facility resident data is offloaded to DASD log data sets. For DASD-only log streams, the log data in local storage buffers is written to DASD log data sets.

If the application disconnects with outstanding asynchronous requests, the disconnect is accepted. Asynchronous requests then complete without notifying the disconnecting application.

Last Disconnection for a System in the Sysplex

System logger offloads all the log data to DASD log stream data sets.

For coupling facility log streams, the coupling facility resident data is offloaded to DASD log data sets. Any coupling facility resources allocated for a coupling facility log stream are released. If a coupling facility structure is being shared among

multiple log streams, the structure space is re-divided evenly among coupling facility log streams with connected applications at that time.

For DASD-only log streams, the log data in local storage buffers is offloaded to DASD log data sets. If there are no other log streams allocated to this coupling facility structure and no failed persistent connections exist, system logger returns the coupling facility space to XES.

Coding a Resource Manager Exit for IXGCONN

A resource manager exit provides a way for system logger to inform a resource manager of write and delete requests so that the resource manager can perform further processing.

When the resource manager connects to the log stream, it specifies:

- The name of the resource manager on the RMNAME parameter.
- The address of the resource manager exit on the RMEXIT parameter.
- User-defined data on the RMDATA parameter.
- The requests that will trigger the resource manager user exit (write and/or delete requests) on the RMEVENTS parameter.

Note that you must be running in supervisor state and a system key to specify the RMNAME, RMEXIT, RMDATA, or RMEVENTS parameters on the IXGCONN request. The resource manager address space must be non-swappable with an AX value of 1, or all invocations of the resource manager exit will fail.

When a write or delete request occurs against the log stream, system logger gives control to the resource manager exit, passing a parameter list. The resource manager exit runs in the resource manager address space.

The resource manager exit is called as follows:

- For a write request, the resource manager exit is called after the write request completes. If staging data sets are in use for the connection, the exit is called after the write to the staging data set completes.
- For a delete request, the resource manager exit is called before the delete request is processed. This allows the resource manager exit to accept, reject, or override the delete request on behalf of the log stream. See “Overriding Delete Requests” on page 26-31.

The resource manager exit is always invoked before the request completion is reported to the system logger application that issued the request.

Information Passed to the Resource Manager Exit.

When the resource manager exit gains control, it receives the following information in the resource manager exit parameter list (RMEPL) mapped by the IXGRMEPL macro:

RMEPLDELETEREQUEST
If on, indicates that the request that gave the resource manager exit control was a delete request.

RMEPLWRITEREQUEST
If on, indicates that the request that gave the resource manager exit control was a write request.

RMEPLGMTTIMESTAMP
Timestamp obtained immediately prior to calling the resource manager.

RMEPLRMNAME
Name of the resource manager the exit belongs to.

RMEPLRMDATA
Data specified in the RMDATA parameter on the resource manager's IXGCONN request, if specified.

RMEPLIDENTIFICATION
System-uniqueidentification of the connection on whose behalf the exit is being called

RMEPLLOGSTREAMNAME
Name of the log stream associated with the resource manager.

RMEPLBLOCKSALLSPECIFIED
If on, BLOCKS=ALL was specified on the IXGDELETE request

RMEPLBLOCKSRANGESPECIFIED
If on, BLOCKS=RANGE was specified on the IXGDELETE request.

RMEPLFORCESPECIFIED
If on, FORCE=YES was specified on the IXGDELET request. This delete request cannot be overridden by the resource manager.

RMEPLDELETEBLOCKID
The block identifier was specified on a IXGDELET BLOCK=RANGE request.

RMEPLDELETEOVERRIDEBLOCKID
Override block identifier. Resource manager places the override block identifier in this variable to override the block identifier issued on the IXGDELET request. If FORCE=YES was specified on the IXGDELET request, the content of this field is ignored and the block identifier specified on IXGDELET is not overridden.

RMEPLADDEDBYTES
The number of bytes that system logger adds to a user's log block for prefix and suffix information. The prefix and suffix areas are not seen in the copy of the user's buffer presented to the resource manager, but if you add together fields RmeplWriteBlockID, RmeplLogDataLength and RmeplAddedBytes, you can calculate the next block identifier that will be assigned for a log block written to the log stream.

RMEPLLOGDATALENGTH
The number of bytes of user log data specified. This is the BLOCKLEN value specified on the IXGWRITE request.

RMEPLWRITEBUFFERPTR
Pointer to the buffer containing the log data written to the log stream if the RMEPLWRITEREQUEST bit is set on.

RMEPLWRITEBLOCKID
Block identifier assigned to the log block written to the log stream.

RMEPLWRITEGMTTIMESTAMP
Timestamp assigned to the log block written to the log stream.

RMEPLWRITELOCALTIMESTAMP
Local time stamp assigned to the log block written to the log stream.

RMEPL_RMEXIT_WORK_AREA
256 byte work area for use by the resource manager.

Environment

The resource manager exit receives control in the following environment:

Authorization:	Supervisor state with PSW key 0
Dispatchable unit mode:	Task or SRB
Cross memory mode:	PASN=resource manager address space, any HASN, any SASN
AMODE:	31-bit
ASC mode:	Primary ASC mode
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held.
Control parameters:	None.

Input Specifications

System logger services pass information to the resource manager exit in registers.

Registers at Entry

When the resource manager exit receives control, the GPRs contain the following information:

Register	Contents
0	Does not contain any information for use by the resource manager exit.
1	Address of a fullword containing the address of the RMEPL
2-13	Do not contain any information for use by the resource manager exit.
14	Return address to system logger services.
15	Entry point address.

When the resource manager exit receives control, the ARs contain no information for use by the resource manager exit.

Return specification

Your exit must return control to the system by branching to the address provided on entry in register 14. Registers 2-13 must contain the same information at output that they did on input.

Programming Considerations

- The resource manager exit is called before write or delete event completion is reported to the unit of work that initiated the IXGWRITE or IXGDELET service request. Any additional processing by the exit that results in thread suspension or affects thread response time should be performed under a different work unit.
- The resource manager exit must be prepared to receive control in either SRB or task mode. The exit should run with an EUT FRR for recovery. While the FRR remains in effect, no SVCs can be issued, no new asynchronous exits are dispatched, and no vector instructions can be executed. See *OS/390 MVS*

Programming: Authorized Assembler Services Reference SET-WTO for information on the SETFRR service.

- All storage obtained by the resource manager exit should be associated with the TCB that owns the cross-memory resources for that address space, whose address is the ASCBXTCB.
- You can access the input parameter data area only while your resource manager exit is running. If you want to save the parameter information for later processing, make a copy of it before your resource manager exit returns control to the system.

Overriding Delete Requests

Your resource manager can override parameters specified on an IXGDELET request by manipulating the RMEPLDELETEOVERRIDEBLOCKID field mapped by the IXGRMEPL mapping macro. The resource manager manipulates the field in the parameter list (RMEPL) passed to the resource manager exit as follows:

- Proceed with the delete operation as requested on the IXGDELET request. This is specified by placing a return code of binary zeros in register 15.
- Do not proceed with the delete operation requested on the IXGDELET request. This results in no log blocks being marked for deletion in the log stream. This is specified by placing a return code of `X'08'` in register 15.
- Override the log block identifier specified on the IXGDELET request with one specified by the resource manager. The overriding log block identifier must be less than or equal to the log block identifier specified on the IXGDELET request. This is specified by placing a return code of `X'04'` in register 15.

If you specify FORCE=YES on a delete request, the resource manager exit is called, but cannot override the delete request.

If you specify AUTODELETE=YES for a log stream and you also manage that log stream with a resource manager and a resource manager exit, note that the automatic deletion processing takes precedence over the delete override processing performed by the resource manager exit. Log data that is deleted by automatic deletion does not trigger the resource manager exit, so the exit cannot override the delete request. IBM recommends AUTODELETE=NO for a log stream managed by a resource manager that needs to override delete requests.

When the Resource Manager Exit Hangs

If the resource manager exit hangs, the system logger application that issued the write or delete request that triggered the exit might not be able to complete. Do the following to resolve the hang:

- Cancel the task that issued the write or delete request.
- Cancel the resource manager's address space.
- Restart the application that issued the write or delete request.
- Correct the hang condition in the resource manager exit.
- Restart the resource manager's address space.

If the resource manager abends and percolates the error back to system logger's recovery environment, the resource manager is disabled. When a resource manager is disabled, the exit for the resource manager is no longer called by write or delete requests against the log stream **on the system where the resource**

manager abended. An ENF 48 event is issued when the resource manager exit is disabled.

IXGWRITE: Writing to a Log Stream

Use the IXGWRITE service to write data from the user's buffer to a log stream. The way system logger processes an IXGWRITE request depends on whether the log stream is coupling facility or DASD-only:

For coupling facility log streams, system logger writes the data to the coupling facility structure space associated with the log stream when an IXGWRITE request is issued. System logger also duplexes the data to either local storage buffers or staging data sets.

For DASD-only log streams, system logger writes the data to interim storage in local storage buffers for the system associated with the log stream. It is also simultaneously duplexed to DASD staging data sets. The safe import point for a DASD-only log stream is changed for every successful write request.

The Log Block Buffer

Before you issue the IXGWRITE service to write your data to the log stream, you must place the data in a buffer to form the log block. This buffer must follow these guidelines:

- The storage key for the buffer, specified on the BUFFKEY parameter, must be one of the following:
  - If the caller is in problem program state, the buffer must be in the same storage key as the caller's PSW key.
  - If the caller is running in supervisor state, the buffer can be in any key (0 through 15).
- The buffer must be either ALET qualified (BUFFALET parameter), or reside in the caller's primary address space.
- The buffer length specified on the BUFFLEN parameter must not exceed the maximum buffer size defined in the LOGR policy for the coupling facility structure.

The format of the data in the log block is entirely the choice of the application; system logger does not have rules about how data looks in a log block.

For each log block written to the log stream, IXGWRITE returns a unique log block identifier which can be used on subsequent IXGDELET and IXGBRWSE requests to search for, delete, read, or set the browse cursor position to that log block.

Ensuring Chronological Sequence of Log Blocks

When an application writes log blocks to a log stream, system logger generates a time stamp for the block, in both local and Greenwich mean time (GMT), showing the time that system logger processed the block. The local time stamp is the local time of the system where the IXGWRITE was issued. Note that local time stamps can repeat because of daylight savings time. In such a case of duplicate time stamps, system logger will return the first block with a matching time stamp that it finds.

Log blocks are placed in the log stream in the order they were received by system logger. System logger generates a GMT timestamp for each log block it receives. Note that the order in which the log blocks are received is not necessarily the same order in which log blocks were written, because when multiple applications write to the same log stream, the log blocks might not be received in the same order that they were written.

An application imbedded timestamp will not affect the order of the log blocks in the log stream. If an application needs to ensure that log blocks are received into the log stream in the order written, **IBM recommends** that applications serialize on the log stream before they write to it.

Applications can optionally request that IXGWRITE return the time stamp that system logger generates for a log block using the TIMESTAMP parameter.

When is Data Committed to the Log Stream?

When you issue the IXGWRITE service to write data to a log stream, you cannot consider the data committed until system logger returns control to the user with a successful return code. Particularly when dealing with a coupling facility log stream, you should never rely on a block ID associated with uncommitted data. For example, consider the following scenario involving a coupling facility log stream:

1. Application 1 issues an IXGWRITE service, and the data is written to the coupling facility.
2. Before the data can be duplexed to staging data sets and control returned to the user, application 2 issues IXGBRWSE for that block of data and successfully reads the data.
3. Now, suppose the coupling facility fails - the data that application 2 read was never committed and is now lost! Application 2 is using log data which no longer exists.
4. At this point, system logger might assign the identical block ID to a different log block and applications 1 and 2, which are trying to share a log, are now out of sync.

For a DASD-only log stream there is less likelihood of losing data, because log data is duplexed automatically to a staging data set. Data is committed when the log data has been written to local storage buffers and duplexed to the staging data set. Note that for a DASD-only data set, log data cannot be browsed until it is committed.

When the Log Stream Coupling Facility Storage Limit Is Reached

An IXGWRITE request will be rejected with return code X'08' and reason code X'0860' when the storage limit for the coupling facility structure associated with a coupling facility log stream is reached. Although the data offload to DASD process generally ensures that data gets written to DASD before the coupling facility structure fills up, this condition can occur due to sudden bursts of activity or when another log stream is activated and defined to the coupling facility. When this happens, system logger will offload data from the coupling facility to DASD immediately. Applications should not issue any further write requests until receiving the ENF event 48 signal indicating that the storage problem is resolved. In the event 48 parameter list mapped by macro IXGENF, bits

IXGENFLogStreamsAvailable and IXGENFLogStreamStorageAvailable will be on and field IXGENFLogStreamNames will contain the names of affected log streams.

When the Staging Data Set Storage Limit is Reached

If the staging data set storage limit for a coupling facility or DASD-only log stream connection is reached, an IXGWRITE request will be rejected with a return code of X'08' and a reason code of X'0865'. When this happens, system logger initiates the offload process to move log data to DASD log data sets and delete corresponding staging data set log data immediately. When offload processing completes, write requests can complete successfully and the staging data set is available to hold duplexed log data again. Applications should not resume issuing write requests until receiving the ENF event indicating that the staging data set storage problem is resolved. In the event 48 parameter list mapped by macro IXGENF, bit IXGENFStagingDsStorageAvailable bit will be on and field IXGENFLogStreamNames will contain the names of the affected log streams.

When the Staging Data Set is Formatting

If the staging data is formatting form initial connect for a system, and IXGWRITE request will be rejected with a return code of X'08' and a reason code of X'0868'. Applications should not resume issuing write requests until receiving the ENF even indicating that the staging data set is available, meaning the format is complete. In the event 48 parameter list mapped by macro IXGENF, indicator IXGENFStagingSsStorageAvailable bit will be on and field IXGENFLofStreamNames will contain the names of the affected log streams.

IXGBRWSE: Browsing/Reading a Log Stream

Use the IXGBRWSE macro to read and browse a log stream for log block information. Applications can:

- Start a browse session and select the initial cursor position - (REQUEST=START)
- Reset the browse cursor position - (REQUEST=RESET)
- Read consecutive log blocks - (REQUEST=READCURSOR)
- Select and read a specific log block - (REQUEST=READBLOCK)
- End a browse session - (REQUEST=END)

IXGBRWSE reads the specified log block into an output buffer specified by the application.

IXGBRWSE Terminology

Before you can read information from the log stream, you start a **browse session** using the REQUEST=START request of IXGBRWSE. A browse session lasts from the time that an application issues IXGBRWSE REQUEST=START until it issues IXGBRWSE REQUEST=END. A log stream can have multiple browse sessions occurring at the same time.

The REQUEST=START request returns a **browse token**, which is a unique 4-byte identifier for a particular browse session. Subsequent IXGBRWSE requests in a browse session use the browse token to identify the session to system logger. Once an application issues the REQUEST=END request, the browse session ends

and system logger will no longer accept IXGBRWSE requests with the browse token for that session.

The **browse cursor** indicates where in the log stream IXGBRWSE will resume browsing on the next request. Each browse session has a browse cursor.

IXGBRWSE Requests

REQUEST=START starts the browse session for the application and sets the browse cursor to the desired starting point. You can specify the browse cursor position for a session using one of the following optional parameters:

- OLDEST - which is the default, starts the browse session at the oldest (earliest) log block.
- YOUNGEST - starts the browse session at the youngest (latest) log block. Note that the record that is the youngest when the browse session starts might no longer be the youngest record at the end of the browse session because of concurrent write activity to the log stream.
- STARTBLOCKID - specifies that the browse session start at a specified log block identifier. The block identifier for a log block is returned by system logger when it is written to the log stream (IXGWRITE) in the field specified by the RETBLOCKID parameter.
- SEARCH - specifies that the browse session start at the log block with the specified time stamp. See “Browsing for a Log Block by Time Stamp” on page 26-36 for details on how IXGBRWSE processes time stamps.

REQUEST=RESET positions the browse cursor to either the oldest or youngest (POSITION=OLDEST or POSITION=YOUNGEST) log block in the log stream.

REQUEST=READBLOCK reads a selected log block in the log stream. You identify the block you want to read by either the block identifier (BLOCKID parameter) or the time stamp (SEARCH parameter). The block identifier for a log block is returned by system logger in the field specified by the RETBLOCKID parameter when the log block is written to the log stream (IXGWRITE).

REQUEST=READCURSOR reads the next oldest or youngest log block in the log stream, depending on the direction specified on the request (DIRECTION=OLDTOYOUNG or YOUNGTOOLD). The block read also depends on the position of the cursor at the time you issue the request.

Applications must take into account that reading a series of consecutive log blocks using REQUEST=READCURSOR might not yield blocks in sequence by local time stamp. This can happen, for example, because of daylight savings time.

Browsing Both Active and Inactive Data

Using the VIEW parameter on IXGBRWSE, you can choose to browse just active log data or both active and inactive log data. **Active data** is data that has not been deleted via IXGDELET. **Inactive data** is data that has been deleted via IXGDELET but is still in the log stream because of a retention period specified in the log stream definition in the LOGR couple data set. See *OS/390 MVS Setting Up a Sysplex* for information about the retention period.

VIEW=ACTIVE Specifies that the browse request browse just active data. VIEW=ACTIVE is the default.

VIEW=ALL Specifies that the browse request browse all data, both active and inactive.

The VIEW parameter can be specified on both the REQUEST=START and REQUEST=RESET requests of IXGBRWSE. For the duration of a browse session, the VIEW specification remains in effect.

Browsing for a Log Block by Time Stamp

System logger generates a time stamp in both local and Greenwich mean time (GMT) for each log block in the log stream. The time stamp is returned in the TIMESTAMP output field when the block is written to the log stream using IXGWRITE. Note that the local time stamp is the local time of the system where the IXGWRITE was issued.

You can use either the local or GMT time stamp on the SEARCH keyword to search for a system logger generated timestamp. You can specify the SEARCH keyword on the following IXGBRWSE requests:

- REQUEST=START, to set the cursor at a particular log block. or REQUEST=READBLOCK (to read a particular log block),
- REQUEST=START, to set the cursor at a particular log block.

When you use a time stamp as a search criteria, IXGBRWSE searches in the oldest-to-youngest direction, searching for a log block with a matching time stamp. If no match is found, IXGBRWSE reads the next latest (younger) time stamp. When you search by time stamp, the search always starts with the oldest log block in the log stream. Searches by time are not sensitive to the current browse cursor position.

See Figure 26-11 on page 26-37 for an example.

If the time stamp specified is older than any time stamp in the log stream, then the oldest time stamp is returned.

If the time stamp specified is younger than any existing time stamps, the request is rejected and the caller receives a return code of X'08' and a reason code of X'0804'.

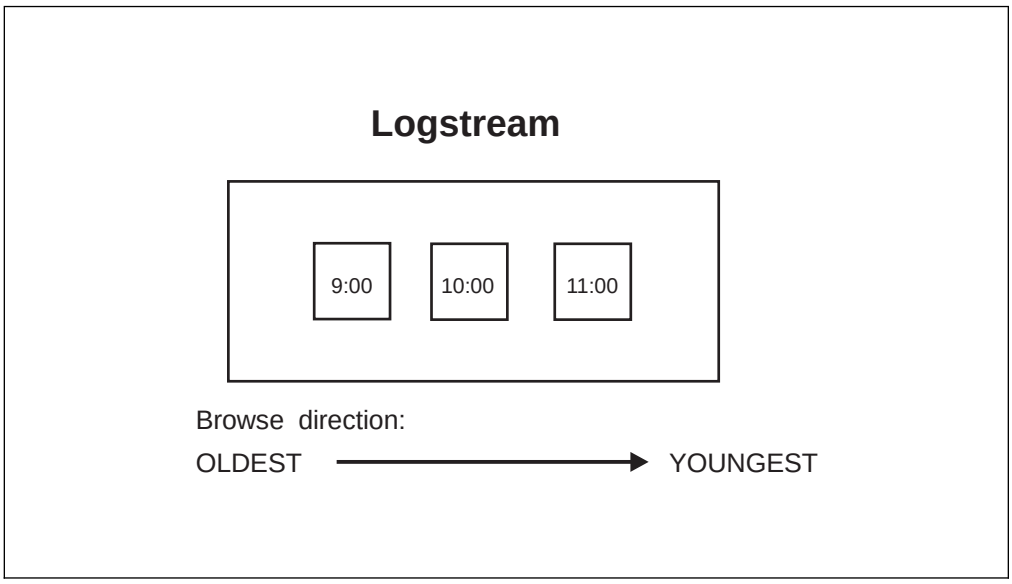


Figure 26-11. Searching for a Log Block by Time

Given the example log stream in Figure 26-11, system logger would do the following:

- If you specify 8:00 on the SEARCH keyword, this is older (earlier) than any log block in the log stream and IXGBRWSE will set the cursor at or returns the oldest time stamp, in this case, 9:00.
- If you specify 10:30 on the SEARCH keyword, IXGBRWSE sets the cursor at or returns the next latest (youngest) log block, 11:00.
- If you specify 12:00 on the SEARCH keyword, this time stamp is younger (later) than any existing log block and IXGBRWSE rejects the request with a return code of X'08' and a reason code of X'0804'.

Using IXGBRWSE and IXGWRITE

If you have applications issuing IXGWRITE and IXGBRWSE requests concurrently for the same coupling facility log stream, ensure that the browse requests are issued only for committed log data. Data is committed to the log stream when system logger returns control to the application following an IXGWRITE request with a successful return code. This is important because data can be lost due to system or coupling facility failure between the time data appears on the structure associated with the log stream and a commit occurs. If you read uncommitted data that is subsequently lost, applications sharing the same log stream can have different versions of log data. See also “When is Data Committed to the Log Stream?” on page 26-33.

Using IXGBRWSE and IXGDELET Requests Together

If you issue IXGDELET and IXGBRWSE requests concurrently, be careful not to delete information before you try to read it. An IXGDELET request can also affect a browse session for a log stream by deleting a requested log block or the log block where the browse cursor is positioned. When an application issues an IXGBRWSE request for log data which has been deleted, the IXGBRWSE request will return non-zero return and reason codes.

Applications might want to provide serialization on the log stream or some other installation protocol to prevent deletes (IXGDELET service) from being performed by other applications on the log stream during a browse session.

IXGDELET: Deleting Log Blocks from a Log Stream

Using the IXGDELET service, you can mark some or all of the log blocks in the log stream for deletion. For a coupling facility log stream, the group of blocks you specify for deletion can reside on both the coupling facility and DASD log data sets. For a DASD-only log stream, the group of blocks you specify for deletion can reside on both the local storage buffers and DASD log data sets. The way system logger processes log data that is marked for deletion depends on the level of the current primary LOGR couple data set for the sysplex and whether a retention period and automatic deletion have been specified for a log stream in the LOGR couple data set. See the system logger chapter in *OS/390 MVS Setting Up a Sysplex* for more information.

Using the BLOCKS parameter

If you specify BLOCKS(ALL) to delete all of the blocks in the log stream, system logger immediately marks as deleted all the blocks that exist at the time of the request. If other applications are writing to the log stream concurrently with the delete request, there might be log blocks in the log stream even after the IXGDELET BLOCKS(ALL) request is processed.

When you want to delete a subset of log blocks, specify BLOCKS(RANGE) and a block identifier on the BLOCKID parameter. System logger marks as deleted **all the log blocks older (written earlier) than the specified log block.** See Figure 26-12 on page 26-39 for an illustration of how BLOCKS(RANGE) works. Note that the block specified in BLOCKID is **not** deleted.

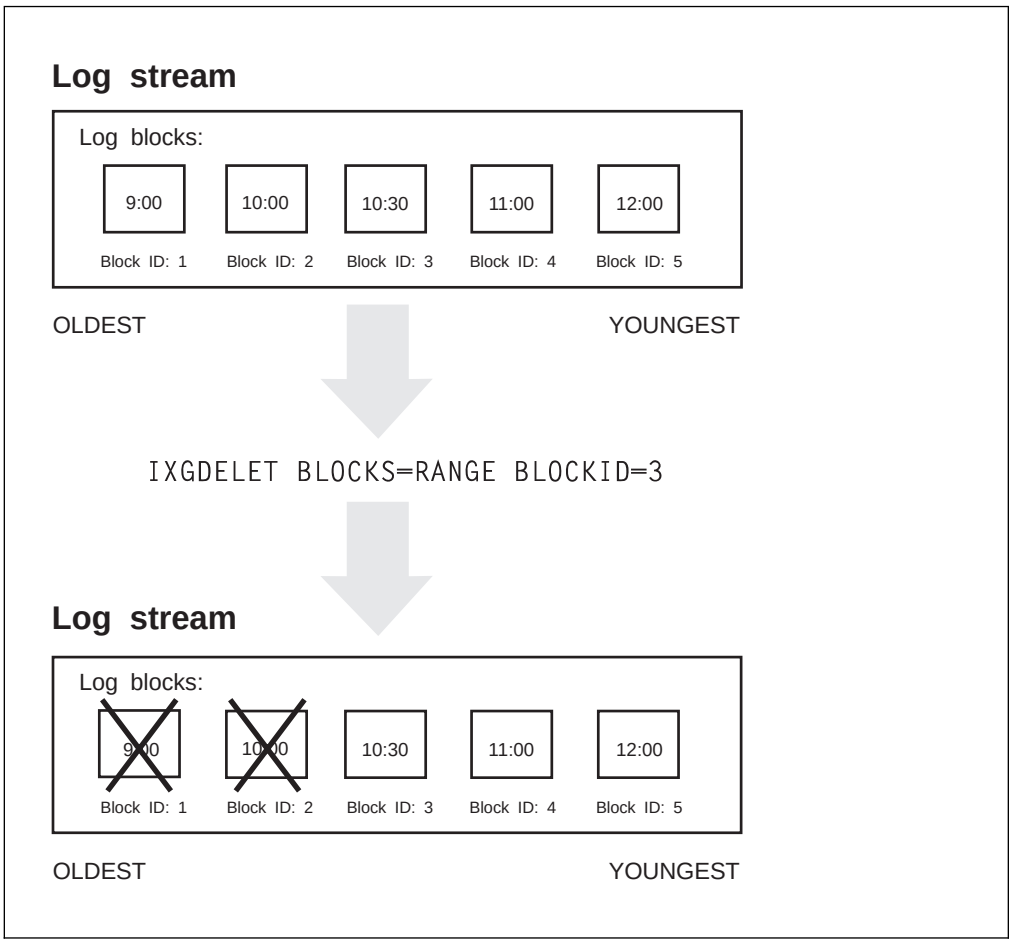


Figure 26-12. Deleting a Range of Log Blocks

Delete Requests and Resource Manager Exit Processing

If you are using a resource manager exit, your resource manager exit can override certain delete requests (see “Coding a Resource Manager Exit for IXGCONN” on page 26-28). You can keep the resource manager exit from overriding a delete request by specifying the FORCE=YES parameter. FORCE=NO, which is the default, allows a resource manager exit to override the delete request. If the resource manager overrides a delete request, system logger returns the overridden block identifier in the OBLOCKID output parameter on IXGDELET.

A resource manager program must connect to the a log stream in order to override delete requests. See “IXGCONN: Connecting to and Disconnecting From a Log Stream” on page 26-22 for more information.

IXGIMPRT: Import Log Blocks

Use the IXGIMPRT service to write data to a log stream, specifying a log block identifier and GMT time stamp for each log block. This service can be used to import copies of log data from one log stream to another, preserving the GMT time stamp and log block identifier assigned when the log block was written to the source log stream. The **source log stream** is the original log stream, the log stream are importing blocks from. The log stream you import blocks to is the **target log stream**.

In order to use the IXGIMPRT service, the connection to the log stream must be an import connection, issued with AUTH=WRITE IMPORTCONNECT=YES. Note that when you specify AUTH=WRITE IMPORTCONNECT=YES for a connection, you cannot issue the IXGWRITE request against the connected log stream. See “IXGCONN: Connecting to and Disconnecting From a Log Stream” on page 26-22 for information about the IMPORTCONNECT parameter.

You must be in task mode to issue the IXGIMPRT service or else the request is rejected.

Making Sure Log Blocks are Imported in Sequence - Understanding Log Block Identifiers

When you import data to a log stream (using IXGIMPRT), the requests are issued with a log block identifier and GMT time stamp identical to the matching log block in the source log stream. The application must make sure to import these log blocks in ascending log block/GMT time stamp order.

For example, if the importing application has log blocks with identifiers 1, 2, and 4 ready to import, the application might need to wait and check for log block 3 before importing 1, 2, and 4 into a log stream. Once log block 4 has been imported, log block 3 can never be imported into the log stream (unless you delete and redefine the log stream). In order to determine whether it is importing log blocks in the correct order, the application must understand the way system logger generates log block identifiers.

The block identifier consists of the logical relative byte address of the log block in relation to the start of the log stream. The first log block written to a log stream is assigned a block identifier of one. Whenever a system logger application writes a log block successfully to the log stream, system logger adds additional control information to the log block. To generate the sysplex-wide unique block identifier, system logger uses:

- The block identifier of the last log block written.
- The length of the current log block (specified by the caller).
- The length of control information (determined by system logger).

The formula is as follows:

```
Length of current log block
           +
length of control information
           +
last log block identifier.
```

How Do I Know What the Length of the Control Information Is?

Applications can ascertain the length of the control information generated by system logger using the IXGQUERY service, which returns the information in a buffer mapped by the IXGQBUF macro (QBUF_CONTROL_INFO_SIZE field).

Example: How Log Block Identifiers are Generated

The following is an example of how log block identifiers are generated:

- The log block identifier generated for the first log block is one.
- The first log block is one hundred bytes. The length of the control information system logger adds to a log block for this log stream is 25 bytes: this information was found using the IXGQUERY service.
- The block identifier for the second log block is generated by adding the first log block identifier to the length of the first log block and the length of the control information: 1+100+25 or 126.
- The 2ND log block is 50 bytes in length, and system logger again added 25 bytes of control information. The block identifier for the third block is 50+25+126 or 201.

Making Sure Log Data is Safe to Import

If your application is using IXGIMPRT to create a duplicate log, copying information from a source to a target log stream, you must make sure that the data is safe to import. See “The Safe Import Point: Using IXGQUERY and IXGIMPRT Together.”

IXGQUERY: Get Information About a Log Stream

Use the IXGQUERY service to retrieve information about a log stream in the sysplex. The information is returned in a buffer mapped by IXGQBUF.

The information returned by IXGQUERY includes:

- Safe import point. See “The Safe Import Point: Using IXGQUERY and IXGIMPRT Together.”
- Control information size: This value shows the number of bytes that system logger adds to each log block written to the log stream.
- Structure version number for coupling facility log streams. See “The Coupling Facility List Structure Version Number” on page 26-44.

  For DASD-only log streams, this value shows the STCK format time stamp that the staging data set for the DASD-only log stream was allocated.

You must be in task mode to issue the IXGQUERY service or else the request is rejected.

See *OS/390 MVS Data Areas, Vol 2 (DCCB-ITTCTE)* for information about the IXGQBUF fields.

The Safe Import Point: Using IXGQUERY and IXGIMPRT Together

If you have an application (a resource manager, for example) that uses the IXGIMPRT service to import log data from a source log stream to a target log stream, creating a duplicate or back-up log stream, you can use IXGQUERY to ascertain the safe import point for a log block before importing it, to make sure the two log streams are synchronized.

See:

- “Coupling Facility Log Streams and the Safe Import Point” on page 26-42.

- “DASD-Only Log Streams and the Safe Import Point” on page 26-44.

Coupling Facility Log Streams and the Safe Import Point

Keeping the log streams synchronized can be particularly difficult when the situation involved coupling facility log streams. If the importing application imports data from a source to a target log stream before it is safely hardened on DASD log data sets or staging data sets, you might get inconsistencies between the source and target log streams. For example, if an importing application involves coupling facility log streams, it is particularly difficult to keep the two log streams synchronized. For example, Figure 26-13 on page 26-43 illustrates the problem. This figure shows a source log stream, SOURCE1, that is written to by applications on two systems, SYS1 and SYS2. SYS1 also contains the coupling facility containing the structure associated with SOURCE1. That means that SYS1 contains a single point of failure and is therefore backed up by a staging data set. SYS2 is a failure independent connection and is not using a staging data set for SOURCE1.

SYS2 has written log blocks 4, 5, and 6 to the log stream coupling facility structure associated with SOURCE1. SYS1 wrote 1, 2, and 3 to the SOURCE1's coupling facility structure. The importing application on SYS3 browses log stream SOURCE1, seeing log blocks 1 through 6 and imports them to log stream TARGET1.

Meanwhile, on SYS1 log blocks 1 and 2 got duplexed to the staging data set, but before 3 could be duplexed, a failure in SYS1 knocked out both the MVS system and the coupling facility. When a rebuild is initiated, only SYS2 can participate to repopulate the log stream. SYS2 repopulates the new structure with the data it wrote (log blocks 4, 5, and 6) and gets blocks 1 and 2 from the staging data set for SYS1. But log block 3 is not repopulated to the rebuilt structure because the data was not committed to the staging data set. Thus, after the rebuild, log stream SOURCE1 contains blocks 1, 2, 4, 5, and 6, while TARGET1 contains 1-6. The source and target log streams are now out of sync.

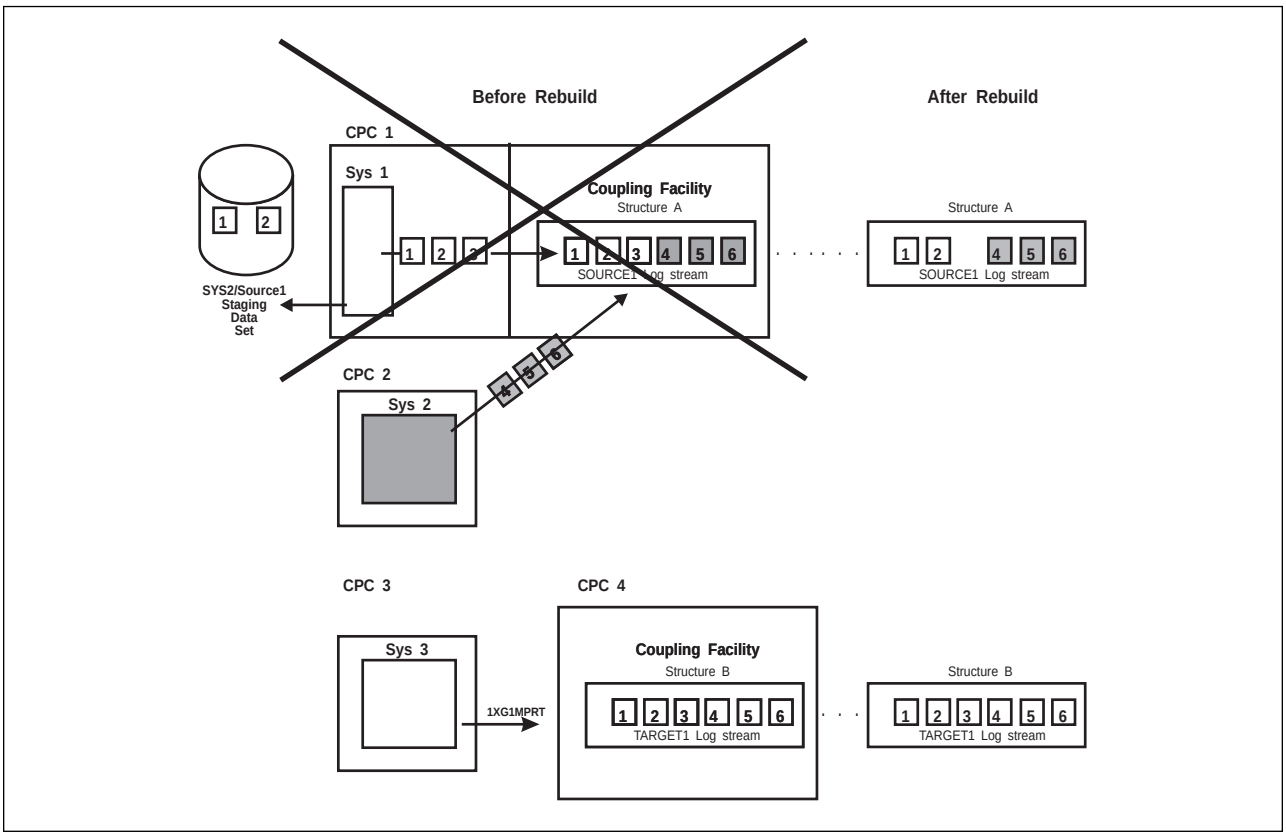


Figure 26-13. How Source and Target Log Streams Can Get Out of Sync

To prevent this problem, system logger provides the safe import point. This helps an application ensure that log data is safe to import. The **safe import point** is the highest log block identifier that is no longer in the coupling facility structure for a log stream. Any blocks with a block identifier less than or equal to the safe import point can be safely imported to the target log stream. In the example shown above, log streams SOURCE1 and TARGET1 are potentially out of sync unless the importing application on SYS3 determines the safe import point before importing the data to the target log stream. An application can determine the safe import by doing one of the following:

- Issue IXGQUERY, which returns the safe import point.
- Listen for ENF event 48 indicating that the safe import point has been updated, (field IXGENFWrOffLoadSafeImportPoint in the parameter list mapped by macro IXGENF contains the latest safe import point.) The ENF is issued when an offload operation completes. Notification that the offload has completed is presented on all systems in the sysplex that have a connection to the log stream.

For example, using the example in Figure 26-13 again, log blocks 1-6 have been written to log stream SOURCE1. Log blocks 1 and 2 have been offloaded to DASD log data sets, so log stream SOURCE1 starts with a safe import point of 2. As before, only blocks 1 and 2 from SYS1 are duplexed to SYS1's staging data set. Log block 3 has not yet been duplexed. The importing application has browsed log stream SOURCE 1 and seen blocks 1-6, but this time, before importing them all, it issues IXGQUERY to find that the safe import point is 2. The resource manager exit imports only log blocks 1 and 2.

When the rebuild occurs, SYS2 repopulates the new structure with log blocks 1, 2, 4, 5, and 6. Again, log block 3 is not repopulated to the rebuilt structure because the data was not committed to the staging data sets. When the browsing application is informed of the rebuild via ENF event (bit IXGENFStrRebuildComplete is on in the parameter list mapped by macro IXGENF) it should then do the following to keep the target log stream synchronized with source log stream SOURCE1:

- Issue IXGQUERY for the new safe import point. Let's say the new safe import point is 6.
- Browse log stream SOURCE1 from the last safe import point it knew about (2) to the new safe import point (6). After the rebuild, SOURCE1 contains 4, 5, and 6.
- Import log blocks 4, 5, and 6 to TARGET1.

The end result is that both SOURCE1 and TARGET1 match, containing log blocks 1, 2, 4, 5, and 6.

Note that an application can also try to change the safe import point by initiating an offload using the IXGOFFLD service.

DASD-Only Log Streams and the Safe Import Point

For a DASD-only log stream data is duplexed to DASD staging data sets as it is written to the log stream. This means that the chance of losing data because of a system failure is far less; There is no lag time between a successful write and hardening of the data on DASD. And of course, a coupling facility failure will not affect a DASD-only log stream. However, it can be helpful for an application to use the IXGQUERY service to ensure that log data is safe to import from a DASD-only source log stream. For a DASD-only log stream, the safe import point is the highest log block identifier that has been successfully written to the log stream. The safe import point for a DASD-only is updated with each successful write, because the log data is simultaneously duplexed to DASD staging data sets. Any log blocks in the source log stream with a block identifier less than or equal to the safe import point can be safely imported to the target log stream. An application can determine the safe import point by doing one of the following:

- Issue IXGQUERY, which returns the safe import point.
- Listen for ENF event 48 indicating that the safe import point has been updated, (field IXGENFWrOffLoadSafeImportPoint in the parameter list mapped by macro IXGENF contains the latest safe import point.) The ENF is issued when an offload operation completes. Notification that the offload has completed is presented on all systems in the sysplex that have a connection to the log stream.

The Coupling Facility List Structure Version Number

The buffer returned by IXGQUERY (IXGQBUF) contains a version number. The meaning varies for DASD-only and coupling facility log streams, but the value is returned in field QBUF_STRUCT_VERSION_NUMBER for both. An alternate name for the field is QBUF_INSTANCE_VERSION_NUMBER.

For a DASD-only log stream, the version number is the time stamp in STCK format when the staging data set for the DASD-only log stream was created.

For a coupling facility log stream, the version number is the coupling facility list structure version number. This value shows the list structure version number of the coupling facility structure assigned to the log stream. A version number is assigned when a structure is initially allocated in the coupling facility and increases every time a structure is rebuilt or reallocated.

Using the Coupling Facility Version Number

The coupling facility list structure version number returned for coupling facility log streams is helpful in situations where a rebuild occurs while the resource manager responsible for gathering data for a log stream was inactive at the time of the rebuild. A browsing application can tell if a rebuild has occurred since it was last connected by checking to see if the coupling facility list structure version number has increased since the last browse session. If a browsing application connects to a log stream and issues IXGQUERY to find that the version number has increased, it should then begin browsing from the last valid safe import point it knew about for the most up to date version of the log stream contents.

For example, imagine an application browses log blocks 1 through 6 in a source log stream in anticipation of importing them. Before importing the log blocks to the target log stream, the application checks for the safe import point using IXGQUERY and imports only blocks 1 and 2. Then, the browsing application is cancelled or fails and disconnects from the log stream, saving the current coupling facility list structure version number and safe import point it knows about (2).

A rebuild then occurs and the new content of the source log stream consists of 1, 2, 4, 5, and 6. The browsing application then reconnects to the log stream and issues IXGQUERY for the new list structure version number and safe import point. Since a rebuild has occurred, the version number will have increased. The browsing application begins browsing again from the last safe import point (2) to the new safe import point (6), and only imports log blocks 4, 5, and 6.

IXGOFFLD: Initiate Offload to DASD Log Data Sets

Use the IXGOFFLD service to initiate an offload of log data to DASD log data sets.

For coupling facility log streams log data is offloaded from the coupling facility structure associated with a log stream to DASD log data sets. Because offloading hardens log data on DASD log data sets, it updates the safe import point for a log stream. You can use IXGOFFLD to initiate an offload on a source log stream to increase the safe import point. This makes more log data available for safe import from the source to the target log stream.

For DASD-only log streams log data is offloaded from local storage buffers to DASD log data sets. Offloading does not update the safe import point, because for DASD-only log streams, when an IXGWRITE service is issued to write data to local storage buffer, the log data is automatically hardened on DASD staging data sets at the same time. For DASD-only log streams, the safe import point is updated after every successful write request.

The number of off-loads initiated via IXGOFFLD are tracked in a field in SMF record 88.

You must be in task mode to issue the IXGOFFLD service or else the request is rejected.

Managing a Target Log Stream: Using IXGIMPRT, IXGOFFLD, and IXGQUERY Together

An application managing a target log stream can monitor the offloads occurring on the source log stream via ENF event 48 writer offload completion events as an indication of when there will be more data to import (IXGIMPRT) to the target log stream. But the application **must also** check the safe import point of the source log stream. The application can do this in one of two ways:

- Listen for ENF event 48, notifying them of a change to the safe import point (field IXGENFWrOffLoadSafeImportPoint in the parameter list mapped by macro IXGENF contains the latest safe import point).
- Issuing the IXGQUERY service (see “The Safe Import Point: Using IXGQUERY and IXGIMPRT Together” on page 26-41) to verify that the data is safe to import to the target log stream.

You must verify the safe import point because it is not always updated for an IXGOFFLD initiated offload even though the safe import point is tied to offloading. This is true because of the way system logger manages log data set control intervals that are not yet full.

IXGUPDAT: Modify Log Stream Control Information

Use the IXGUPDAT service to modify the GMT time stamp control information for a log stream. Once modified, the next log block written to the log stream will be assigned a GMT time stamp equal to or greater than the one specified on the IXGUPDAT request.

For coupling facility log streams, the time stamp for a log stream maintained in the coupling facility list controls.

for DASD-only log streams the time stamp for a log stream maintained in the local control blocks for the system associated with the log stream and the staging data set control records.

On the IXGUPDAT service, you must supply a time stamp that is equal to or greater than the current time stamp maintained in the Log Stream Control information. You can obtain the current time stamp for the log stream by issuing the IXGQUERY service.

You must be in task mode to issue the IXGUPDAT service or else the request is rejected. The program issuing IXGUPDAT must also have specified AUTH=WRITE on the IXGCONN request to connect to the log stream to issue this request.

Why Would I Need to Change a Time Stamp on a Log Stream? The IXGUPDAT service is useful mainly to coupling facility log streams. If an application writes to multiple coupling facility log streams at the time of a failure necessitating recovery processing, database recovery processing truncates the log streams to the latest point consistent across all the log streams. This can mean a loss of data. For example, let's say an application writes to two log streams. In log stream 1, the last log block written was at 9 A.M.. In log stream 2, the last log block was written at 9:15 A.M.. The recovery process logically truncates log stream 2 data to 9 A.M., to be consistent with log stream 1 for recovery. This means that the data written to log stream 2 between 9:00 and 9:15 is lost.

In this situation, you would use IXGUPDAT to update the time stamp for log stream 1 to 9:15 or greater so that all data for both log streams can be recovered. All data written to the two log streams up until 9:15 A.M can then be recovered. IXGUPDAT lets you advance the time stamp for a log stream or streams to the latest time among all the log streams. This ensures that log streams have a consistent time stamp and include all the data written to any of the log streams.

Rebuilds and IXGUPDAT processing

For a coupling facility log stream, when a rebuild occurs after you update a time stamp using IXGUPDAT, the time stamp for the log stream will be set for the time of the last log block written and recovered to the log stream. This will be the last log block written, unless rebuild processing indicates a loss of data condition for the log stream.

If the rebuild occurs after one or more successful write requests have occurred since the IXGUPDAT request, the time stamp is reset for the last post-IXGUPDAT written and recovered log block time stamp. If however, the rebuild occurs before any write requests to the log stream since the IXGUPDAT request, the time stamp for the log stream reverts to the last pre-IXGUPDAT written and recovered write request.

Setting Up the System Logger Configuration

- To set up a system logger configuration for a logging function or application, see see the chapter on planning for system logger functions in *OS/390 MVS Setting Up a Sysplex*.
- For system logger applications, **IBM recommends** that you use ENF event code 48 and write an ENF event code 48 listen exit. See “Writing an ENF Event 48 Listen Exit.”

Writing an ENF Event 48 Listen Exit

Before activating system logger applications that write to the log stream, you should set up an exit to listen for and analyze system logger status information broadcast by ENF as event code 48. Note that your application must be supervisor state, system key to set up an ENF listen exit. Event 48 includes status information about log streams, the system logger component, and log stream coupling facility structures. System logger applications can use the ENF event 48 listen exit to monitor status and changes.

Some events are single system in scope, while others are broadcast to all the systems in the sysplex.

Each application registers interest in ENF event code 48 signals using the ENFREQ macro (see “Using ENF Event Code 48 in System Logger Applications” on page 26-19). When an application is notified via a return and reason code of a problem, such as a coupling facility structure rebuild in progress, system logger address services unavailable, or loss of connectivity to a coupling facility structure, the application can then listen for an event 48 signal signifying that the problem has been resolved. Note that a program must be authorized (supervisor state) to use the ENFREQ service.

Once the application registers interest in event 48 system logger events, system logger passes the event 48 parameter list containing information about the event to the listen exit. The parameter list is mapped by macro IXGENF, see *OS/390 MVS Data Areas, Vol 2 (DCCB-ITTCTE)*. The parameter list contains the specific code 48 events, the reasons associated with them, specific information for some reasons, and a section of information about log streams and coupling facility structures affected by the event.

The ENF event code 48 events are:

- System logger address space and services have become available.
- System logger address space and services are not available for the life of this IPL.
- Log streams associated with the coupling facility structure specified in this parameter list have become available. The event reasons for this event are:
  - A coupling facility structure rebuild in progress has completed.
  - The coupling facility structure is no longer too full to accommodate more log data.
  - A staging data set full condition has ended.
- Log streams associated with the coupling facility structure specified in this parameter list are not available. The event reasons for this event are:
  - A rebuild has been started for the coupling facility structure.
  - A rebuild for the coupling facility structure has failed. The specific reasons for this condition in the parameter list are:
    - The system lost connectivity to the coupling facility structure.
    - The coupling facility structure failed.
- A change in the status of resources for a log stream has occurred. The event reason for this event is possible loss of log stream data.
- A change in the coupling facility resources available to system logger has occurred. An event reason for this event indicates that ENF event 35 was received to report the change. If the change affects a specific coupling facility structure, the name of the structure is specified in the specific information section of the parameter list.
- A successful connect to or disconnect from a log stream has occurred. The scope of this event is multi-system; each system in the sysplex is notified of this event. The structure version value for DASD-only log streams will be the STCK value for when the log stream staging data set was allocated.
- A log stream definition in the LOGR couple data set has been created. The scope of this event is multi-system; each system in the sysplex is notified of this event.
  - The log stream created is a DASD-only log stream.
    - For DASD-only log streams, an indicator in the parameter list will be set on (IXGENFINVENTORYDASDONLYYES) and most structure related fields will be set to zero.
  - A log stream definition has been deleted from the LOGR couple data set. The scope of this event is multi-system; each system in the sysplex is notified of this event.

– A log stream definition in the LOGR couple data set has been updated. The scope of this event is multi-system; each system in the sysplex is notified of this event.

– System logger completed offload processing for a log stream. All systems with an active connection to the log stream for which the offload was done are notified of the event.

– The resource manager associated with a log stream is disabled because of an abend. The system where the resource manager is disabled is notified.

The listen exit should **only** be used to analyze the IXGENF parameter list for event code 48 to see whether the particular event applies to the particular application or connector. IBM recommends that the listen exit then communicate with the application about the event and let the application react or take any necessary actions, such as stop issuing services, re-IPL, and so forth. This is recommended because:

– The application or connector can determine whether the event actually affects them before taking any action. All connectors are informed of event 48 events, whether they affect their particular log stream or coupling facility structure or not.

– It ensures that the action needed to address an event will be coordinated with the application program since the listen exit must be in SRB mode while the application is in task mode. If you code the exit to actually act on event code 48 status events, the exit might tie up system logger resources.

Reading Data From Log Streams in Data Set Format

There are two ways that you can read data from log streams:

- Write a new application to read data that supports the log stream data format using the system logger services, particularly IXGBRWSE. See “IXGBRWSE: Browsing/Reading a Log Stream” on page 26-34.
- Use the LOGR subsystem to access log stream data in data set format for existing applications that need to read log data but do not support the log stream format.

This section describes how to use the LOGR subsystem to access log stream data in data set format. For example, you might have existing applications that read logrec data in data set format. The LOGR subsystem interface allows you to use your existing applications to read log stream output without having to re-write them.

Is My Application Eligible for the LOGR Subsystem?

You can use the LOGR subsystem to access log stream data for programs that conform to the following:

- **Use only BSAM or QSAM access methods**.
- **Use only record lengths allowed by QSAM or BSAM in the log stream**.

  QSAM and BSAM access methods support up to approximately 32K record sizes while a log stream can use a record size up to 64K-4. Make sure that the log stream uses a record size compatible with the access method if you want to use the LOGR subsystem to read log stream data. See *DFSMS/MVS Macro Instructions for Data Sets* for more information about access methods.

- **Use only the following macros to access data:**
  - DCB
  - DCBE
  - RDJFCB
  - OPEN
  - GET (QSAM)
  - READ (BSAM)
  - CHECK (BSAM)
  - TRUNC (BSAM)
  - SYNADAF
  - SYNADRLS
  - CLOSE

  Your program is **NOT** eligible for use with the LOGR subsystem if it uses the NOTE, POINT, and CNTRL macros.

Preparing to use the LOGR Subsystem

If you want use the LOGR subsystem to read log stream data in data set format, make sure of the following:

- The LOGR subsystem must be activated on each system where you expect to use it.

  Note that you cannot use either the SETSSI ACTIVATE or SETSSI DEACTIVATE command to activate or deactivate the LOGR subsystem. See the chapter on system logger functions in *OS/390 MVS Setting Up a Sysplex*.

  Each system must have LOGR defined as a subsystem on the SUBSYS statement in the job that runs the subsystem.
- The log stream owner must supply an exit that generates a view of its log data. You can use this to ensure that the new application program is written with the correct data format in mind. See *OS/390 MVS Programming: Assembler Services Guide*.
- The system where the new log data reading application runs on must be in the same sysplex as the log stream.
- The application must have the appropriate SAF authority (READ/WRITE) to the log stream, SAF CLASS(LOGSTRM) RESOURCE(*log_stream_name*).

  If there is no log stream class defined to a security product such as RACF, no access checking is performed.
- Make sure the LOGR subsystem is activated.

References:

- See the chapter on planning for system logger functions in *OS/390 MVS Setting Up a Sysplex*.
- See *OS/390 MVS Initialization and Tuning Reference* for information about the IEFSSNxx parmlib member.
- See *OS/390 MVS System Commands* for information about the SETSSI command.
- See *OS/390 MVS Installation Exits* for information about the LOGR subsystem exit.

Using the LOGR Subsystem

Do the following to use the LOGR subsystem:

1. Obtain the following information from the log stream owner:
   - Name of the log stream.
   - Name of the exit routine.
2. Add the exit routine and log stream data to the JCL used to invoke the application using the SUBSYS statement. See “JCL for the LOGR Subsystem.”
3. Make sure the LOGR subsystem has been activated. See “Preparing to use the LOGR Subsystem” on page 26-50.

JCL for the LOGR Subsystem

Add the exit routine and log stream data to the JCL used to invoke the application using the SUBSYS statement. The format and explanation of the SUBSYS statement for the LOGR subsystem is as follows:

```
//ddname   DD  DSNAME=log.stream.name,
//           SUBSYS=(LOGR[,exit_routine_name][,'SUBSYS-options1'][,'SUBSYS-options2'])

where:

SUBSYS-options1: [FROM={({[yyyy/ddd][,hh:mm[:ss]]})  | OLDEST}]
[TO={({[yyyy/ddd][,hh:mm[:ss]]})  | YOUNGEST}]
[,DURATION=(nnnn,HOURS)]
[,GMT|LOCAL]

SUBSYS-options2: defined by the log stream owner
```

Figure 26-14. Log Stream SUBSYS Data Set Specification

Note: Quotation marks around keywords are required when parentheses, commas, equal signs or blank characters are used within the SUBSYS keyword.

Other DD keywords will be validated, if specified, but will be ignored in the LOGR subsystem processing.

DSNAME=*log.stream.name*
Specifies the name of the log stream to read. The name can be 1 to 26 characters in a data set name format.

SUBSYS=(LOGR[,*exit_routine_name***][,***'SUBSYS-options1'***][,***'SUBSYS-options2'***])**
Specifies that processing of this DD is to be handled by the LOGR subsystem.

The *exit_routine_name* is the second positional parameter and specifies the name of the exit routine to receive control from the LOGR subsystem. If the *exit_routine_name* parameter is not specified (null), the default log stream subsystem exit routine, IXGSEXIT, will be used. To access records from the logrec log stream, specify IFBSEXIT.

SUBSYS-options1
Specifies options meaningful to all exit routines:

FROM=*starting_time*

Indicates the starting time of the first log stream block to be processed. The first block will be the one with a time stamp later than or equal to the specified time.

OLDEST

Indicates the first block read will be the oldest block on the log stream. OLDEST is the default.

yyyy/ddd

Specifies the start date. If the date is omitted, the current date is assumed.

yyyy is a four-digit year number and *ddd* is a three-digit day number from 001 through 366 (366 is valid only on leap years). For example, code February 20, 2000 as 2000/051, and code December 31, 1996 as 1996/366.

hh:mm[:ss]

Specifies the start time. If the time is omitted, the first block written after midnight will be used.

hh is a two digit hour number from 00 to 23, *mm* is a two digit minute number from 00 to 59, and *ss* is a two digit second number from 00 to 59. The seconds field and associated : delimiter can be omitted if not required by the log stream owner.

The FROM keyword is mutually exclusive with the DURATION keyword.

TO=*ending_time*

Indicates the ending time of the last log stream block to be processed. The last block will be the one with a time stamp earlier than or equal to the specified time.

YOUNGEST

Indicates the last block read will be the youngest block on the log stream at the time the allocation for the DD occurs. YOUNGEST is the default.

yyyy/ddd

Specifies the end date. If the date is omitted, the current date is assumed.

yyyy is a four-digit year number and *ddd* is a three-digit day number from 001 through 366 (366 is valid only on leap years). For example, code March 7, 2001 as 2001/066, and code November 12, 2000 as 2000/317.

hh:mm[:ss]

Specifies the end time. If the time is omitted, the last block written before midnight will be used. If the end date is the same as the current day, then the youngest block on the log stream at the time the allocation for the DD occurs will be used.

hh is a two digit hour number from 00 to 23, *mm* is a two digit minute number from 00 to 59, and *ss* is a two digit second number from 00 to 59. The seconds field and associated : delimiter can be omitted if not required by the log stream owner.

The TO keyword is mutually exclusive with the DURATION keyword.

Note: If the value specified for the FROM keyword is greater than the value specified for the TO keyword, the system ends the jobstep with a JCL error.

DURATION=(*nnnn*,HOURS)
Specifies which blocks are to be processed. Each *n* is a numeric from 0 to 9. *(nnnn,HOURS)* requests the blocks for the "last nnnn hours" up to the youngest block be processed. The "last nnnn hours" are calculated from the current time of the allocation for the DD.

The first block will be the one with a time stamp greater than or equal to the calculated start time. The last block read will be the youngest block on the log stream at the time the allocation for the DD occurs.

The DURATION keyword is mutually exclusive with the TO and the FROM keywords.

GMT|LOCAL
Specifies whether the time is local time (based on the time zone offset at the time the log was written) or GMT time. GMT is the default.

Along with the above general parameters that can be specified for a log stream subsystem data set, system logger provides additional parameters in the *SUBSYS-options2* specifications. See the section on obtaining records from the logrec log stream in *OS/390 MVS Diagnosis: Tools and Service Aids*.

When Things Go Wrong - Recovery Scenarios for System Logger

The following section describes some of the failures that can affect system logger applications and the action taken by system logger in response.

System logger performs recovery differently for DASD-only versus coupling facility log streams. Recovery for DASD-only log streams need to be done by the application. Therefore, most of the information in this section applies to coupling facility log streams only. For DASD-only log streams, see “Recovery Performed for DASD-Only Log Streams” on page 26-54. Other recovery information pertinent to DASD-only log streams are noted under each topic below.

For many of the failures, an application can listen for an ENF 48 event to find out when problems are resolved or resources are available again.

When a System Logger Application Fails

If a system logger application fails while holding active connections to one or more log streams, system logger automatically disconnects the application from the log streams.

If the connection was the last connection to a log stream from a system, all log data written by that system to the log stream is offloaded to DASD log data sets.

When an MVS System or Sysplex Fails

This section applies to coupling facility log streams only; for DASD-only log streams, see “Recovery Performed for DASD-Only Log Streams.”

When a system fails, system logger tries to safeguard all the coupling facility log data for the failed system by offloading it to DASD log data sets so that it is on a persistent media.

Recovery processing for the failing system is done by a **peer connector**, which is another system in the sysplex with a connection to a coupling facility structure that the failing system was also connected to. Note that a peer connector need only be connected to the same coupling facility structure, not the same log stream. See the chapter on planning for system logger functions in *OS/390 MVS Setting Up a Sysplex* for more information.

When all the systems in a sysplex fail, there are no peer connectors to perform the recovery processing for the failing systems, which would consist of offloading the coupling facility data for log streams to DASD log data sets. Coupling facility resident log data continues to exist. Further recovery processing depends on whether or not the coupling facility also failed.

Recovery Performed for DASD-Only Log Streams

Like a coupling facility log stream, a DASD-only log stream is subject to system or system logger failure. A DASD-only log stream is not subject, however, to coupling facility or structure failures. When a failure occurs involving a DASD-only log stream, system logger releases the exclusive ENQ on the log stream name serializing the log stream for one system. No system-level or peer recovery is performed for a DASD-only log stream after a failure or as part of system initialization. System logger does not perform system-level recovery for a DASD-only log stream because data is already safeguarded on DASD staging data sets, a non-volatile medium. For a DASD-only log stream, offload of log data to DASD log data sets is not done as part of recovery processing for the same reason - log data is already on DASD staging data sets. Peer recovery is both unnecessary and not possible for a DASD-only log stream, because there are no peer systems connected to the log stream.

Recovery for a DASD-only log stream only takes place when an application reconnects to the log stream. As part of connect processing, system logger reads log data from the staging data set (associated with the last connection to the log stream) into the local storage buffers of the current connecting system. This allows the application to control recovery, by selecting which system they wish to have reconnect to the log stream and when. Note that for another system to connect to the log stream and perform recovery, the staging data sets must reside on devices accessible by both systems.

When the System Logger Address Space Fails

This section applies to both coupling facility and DASD-only log streams.

If the system logger address space fails, any system logger requests from the system where the system logger component failed are rejected. See *OS/390 MVS Programming: Assembler Services Reference* for information on system logger services.

When the Coupling Facility Structure Fails

This section applies to coupling facility log streams only.

The following coupling facility problems can occur, resulting in rebuild processing for the structure:

- Damage to or failure of the coupling facility structure.
- Loss of connectivity to a coupling facility.
- A coupling facility becomes volatile.

For complete information on rebuild processing, see *OS/390 MVS Programming: Sysplex Services Guide*.

Damage to or Failure of the Coupling Facility Structure

If a coupling facility fails or is damaged, all systems connected to the coupling facility structure detect the failure. The first system whose system logger component detects the failure initiates the structure rebuild process. The structure rebuild process results in the recovery of one or more of the affected coupling facility structure's log streams. All the systems in the sysplex that are connected to the list structure participate in the process of rebuilding the log streams in a new coupling facility list structure.

When the rebuild starts, system logger issues an event 48 signal to inform applications that the rebuild is starting and that the log streams associated to the coupling facility structure that is being rebuilt are not available. Bits IXGENFStrRebuildStart and IXGENFLogStreamsNotAvailable are on in the event 48 parameter list mapped by macro IXGENF.

While the rebuild is in progress, system logger rejects any system logger service requests against the log stream. Applications must listen for another ENF event 48 to learn the status of the log stream after rebuild processing is complete. The status will be one of the following:

- The structure rebuild has completed successfully, the coupling facility structure and associated log streams are available, and system logger requests will be accepted. Bits IXGENFLogStreamsAvailable and IXGENFStrRebuildComplete are on in the event 48 parameter list mapped by macro IXGENF.
- The structure rebuild was unsuccessful and connection to the structure is not possible because the structure is in a failed state. Log data still resides in staging data sets if they are used to duplex the log data for the log stream. If staging data sets were not used, the data persists in the local storage buffers on each system. All system logger service requests against the log streams will be rejected. Bits IXGENFLogStreamsNotAvailable, IXGENFStrRebuildFailed, and IXGENFRebuildFailStrFail are on in the event 48 parameter list mapped by macro IXGENF.

  In this case, applications connected to the affected log streams must wait for the structure to be rebuilt successfully and the system to issue an ENF 48 event to indicate that the log streams are available.

Loss of Connectivity to the Coupling Facility Structure

If a system loses connectivity to the coupling facility structure due to a hardware link failure, all the systems connected to the log streams associated with the coupling facility detect the failure.

Then, based on the rebuild threshold specified, if any, in the structure definition in the CFRM policy, the system that lost connectivity may initiate a rebuild for the structure.

If a rebuild is initiated, the event 48 parameter list mapped by macro IXGENF has bits IXGENFLogStreamsNotAvailable, and IXGENFStrRebuildStart, on, and field IXGENFStrName contains the name of the coupling facility structure affected. System logger rejects logger service requests issued during the rebuild process.

If XES cannot allocate a new coupling facility that all the systems affected can connect to, system logger does one of the following, depending on whether the system or systems that cannot connect to the new coupling facility structure were using staging data sets:

- If the system was using staging data sets, the rebuild process continues and the coupling facility log data for the system is recovered from the staging data sets.
- If the system was **not** using staging data sets, the rebuild process is stopped. The systems go back to using the source structure.

  The systems that do not have connectivity to the old coupling facility structure issue an ENF 48 event indicating that they do not have connectivity to the log stream.

  The systems that can connect to the source structure issue an ENF 48 event indicating that the log stream is available to that system and can resume use of the log stream.

  The installation should either update the CFRM to make the new coupling facility structure available to all the systems or else fix the hardware link problem and then have the operator initiate a rebuild for the structure so that all the original systems will have connectivity.

Applications must listen for another ENF event 48 to learn the status of the log stream after rebuild processing is complete. The status will be one of the following:

- The structure rebuild has completed successfully, the coupling facility structure and associated log streams are available, and system logger requests will be accepted. Bits IXGENFLogStreamsAvailable and IXGENFStrRebuildComplete are on in the event 48 parameter list mapped by macro IXGENF.
- The structure rebuild was unsuccessful. Connections to the structure are not available because the system receiving the ENF event code has lost connectivity to the structure.

  All system logger service requests against the log streams will be rejected. Bits IXGENFLogStreamsNotAvailable, IXGENFStrRebuildFailed, and IXGENFRebuildFailLossConn are on in the event 48 parameter list mapped by macro IXGENF.

A Coupling Facility Becomes Volatile

If a coupling facility changes to the volatile state, the system logger on each system using the coupling facility structure is notified. A dynamic rebuild of the structure is initiated so that the log data can be moved to a non-volatile coupling facility. During rebuild processing, system logger rejects any logger service requests.

If there is not a structure available in a non-volatile coupling facility, system logger will still rebuild the data on a new volatile coupling facility. System logger may then change the way it duplexes coupling facility data since the volatile coupling facility constitutes a single point of failure:

- For log streams defined with STG_DUPLEX=YES, system logger will begin duplexing data to staging data sets, if they were not already in use.
- For log streams defined with STG_DUPLEX=NO, system logger will keep on duplexing data to local storage buffers on each system.

When the Coupling Facility Space for a Log Stream Becomes Full

This section applies to coupling facility log streams only.

Ordinarily, system logger offloads coupling facility resident log stream data to DASD log data sets before the coupling facility storage allocated to the log stream is fully utilized. Occasionally however, the coupling facility storage allocated to a log stream reaches 100% utilization, for reasons such as a sudden burst of logging activity or because your coupling facility is sized too small for the volume of log data.

Applications are notified of a filled log stream by system logger service return code of 8 and a reason code of IXGRsnCodeCFLogStreamStorFull (X'0860'). System logger will not accept write requests until offload processing can be completed and applications receive an ENF signal with bit IXGENFStructureNotFull on.

When a Staging Data Set Becomes Full

This section applies to both coupling facility and DASD-only log streams.

The staging data sets for each system should not fill up. If they do, you probably have them sized too small for your volume of log data and should enlarge them.

When a DASD staging data set becomes full, IXGWRITE requests from the system with a staging data set full condition will be rejected with a return code of 8 and a reason code of IXGRsnCodeStagingDsFull (X'0865').

System logger will immediately begin offloading log data to DASD log data sets. If your staging data set is too small, you may find that offloading occurs very frequently.

System logger will not accept write requests from the system associated with the staging data set until offload processing can be completed and applications receive an ENF signal indicating that the staging data set full condition has ended.

When a Log Stream is Damaged

This section applies to both coupling facility and DASD-only log streams. damaged when it cannot recover log data from either DASD staging data sets or the local storage buffers after a system, sysplex, or coupling facility failure. Applications are notified of the damage by the following means:

- When they issue IXGCONN, they receive return code, X'04', reason code, IXGRsnCodePossibleLossOfData
- When they issue IXGBROWSE, they receive return code, X'04', reason code, IXGRsnCodeWarningGap
- Their listen exit detects ENF event 48.

Applications should do the following to respond to a damaged log stream condition, depending on how critical a data loss is to the application:

- Applications that absolutely cannot tolerate any data loss whatsoever, some short term transaction logs for example, should stop issuing system logger services to the affected log stream, disconnect from the log stream, perform recovery for the application, and then reconnect to a new log stream.
- Applications that can tolerate some data loss, such as archive logs that do not read a great deal of data from the log stream, may be able to continue using the log stream. See “How System Logger Handles Gaps in the Log Stream” on page 26-15 for a summary of the results of reading a damaged log stream.

When DASD Log Data Set Space Fills

This step applies to both coupling facility and DASD-only log streams.

The number of DASD log data sets available for log streams in a sysplex depends on whether you use the default (168 per log stream) or have provided additional directory extents in the LOGR couple data set.

System logger monitors usage of the available log stream directory space, notifying you as follows if you start to run out of space:

- If you are using the DSEXTENT parameter in the LOGR couple data set system logger issues messages IXG261E and IXG262A indicating that usage of directory extents is over 85% and 95% respectively.
- If you are using the default number of log data sets allowed for a log stream (168), system logger issues message IXG257I indicating that the data set directory for the log stream is over 90% full.

If you have run out of log stream directory space, offloads may fail. When this occurs, system logger issues message IXG301I. Offload processing for the log stream cannot complete until more log stream directory space or directory extents are made available. If the last disconnect to a log stream occurs and the offload cannot complete successfully, the log stream is in a failed state. In this case, the log stream is considered 'in use' and there may be a failed-persistent connection to a structure associated with the log stream.

You can make more directory space available for a log stream in one of the ways below. Use the IXCMIAPU utility to run a report of the log stream definitions in the LOGR couple data set to help you with this step. The LIST LOGSTREAM NAME(*) DETAIL(YES) statement outputs information showing which log streams might be

using large numbers of data sets and directory extents. See *LOGR Parameters for Administrative Data Facility* in *OS/390 MVS Setting Up a Sysplex* for more information about IXCMIAPU.

- Format another set of LOGR couple data sets with a higher DSEXTENT value and bringing them into the sysplex as the active primary and alternate LOGR couple data sets. You must have a LOGR couple data set formatted at the OS/390 Release 3 level or above to use the DSEXTENT parameter. See *Appendix B: Administrative Data Utility* in *OS/390 MVS Setting Up a Sysplex* for the IXCL1DSU utility and the DSEXTENT parameter.
- Free directory extents currently in use in one of the following ways:
  - Use a program that issues the IXGDELET service to delete enough data from the log stream to free up space in the log stream data set directory.

    Some products provide a program to delete log stream data. See *Deleting Log Data and Log Data Sets* in *OS/390 MVS Setting Up a Sysplex* for information on deletion programs provided for IBM products. See the documentation for the product to see if it provides a deletion program.
  - Delete log stream definitions from the LOGR couple data set.

    Identify and delete the definitions for unused or unnecessary log streams. This will free the directory space associated with the log streams, which may free up directory extents for use by other log streams.

  Note: Deleting DASD log data sets using a non-system logger method will not work because system logger will still count the data sets toward the data set directory entry limit. You cannot, for example:

  - Use a TSO/E DELETE command to delete a log data set.
  - Use DFHSM to migrate log stream data sets to tape.

When Unrecoverable DASD I/O Errors Occur

This section applies to both coupling facility and DASD-only log streams, with differences noted.

DASD I/O errors may occur against either log data sets or staging data sets. System logger tries to recover from the error, but if it cannot, the error is characterized as an **unrecoverable I/O error**. See the following:

- "When Unrecoverable DASD I/O Errors Occur During Offload"
- "When Staging Data Set Unrecoverable DASD I/O Errors Occur" on page 26-60

When Unrecoverable DASD I/O Errors Occur During Offload

DASD I/O errors may occur during offload processing, while log data is being written to DASD log data sets. When this happens, system logger tries to recover by closing the current log data set and allocating a new one. If this process fails, the I/O error is characterized as an unrecoverable I/O error.

In the case of unrecoverable I/O errors, system logger will accept subsequent IXGWRITE requests as follows:

- For a coupling facility log stream, system logger will accept IXGWRITE requests if the log stream is connected to a coupling facility where there is still

room for log data. If the coupling facility is full or no coupling facility exists, system logger rejects IXGWRITE requests.

- For a DASD-only log stream, system logger will accept IXGWRITE requests until the staging data set for the system writing to the log stream is filled.

IXGBRWSE and IXGDELET requests may continue to work. I/O errors encountered in the process of completing these requests are reported to the application in return and reason codes.

To correct an unrecoverable I/O problem, delete the log stream definition in the LOGR policy and redefine it with different log data set attributes, such as LS_DATACLAS, in order to get the log stream data set allocated in a usable location.

When Staging Data Set Unrecoverable DASD I/O Errors Occur

DASD I/O errors may occur when log data is being duplexed to DASD staging data sets. When this occurs, system logger tries to recover by doing the following:

1. Offload current log data to DASD log data sets.
2. Delete and unallocate the staging data set.
3. Re-allocate a new instance of the staging data set.

In the meantime, system logger continues to accept write and other requests against the log stream.

If system logger cannot re-allocate a new staging data set, the I/O error is characterized as unrecoverable. In the case of an unrecoverable staging data set I/O error, system logger does the following:

- **For a coupling facility based log stream**, system logger switches the duplexing mode to duplex log data to local storage buffers. The system issues message IXG255I indicating that the duplexing mode has changed. Normal system logger processing continues. The log stream may be more vulnerable to data loss due to system, sysplex, or coupling facility failure.
- **For a DASD-only log stream**, system logger disconnects connectors from the log stream. The system issues message IXG216I to indicate that connectors are being disconnected because a staging data set could not be allocated.

When A Resource Manager Fails

When a resource manager percolates to the recovery environment of system logger, it is disabled with abend X'x22', regardless of whether it had the opportunity to retry. When this happens, the resource manager must disconnect from the log stream and then reconnect in order to activate the resource manager exit.

When a resource manager is disabled, an ENF 48 event is issued on the system where it is disabled. If an SDWA is available to system logger's recovery routine, the abend code is included in the ENF parameter list mapped by macro IXGENF.

The resource manager should examine the abend code and decide whether to disconnect and terminate or to disconnect and reconnect to the log stream.

See “Writing an ENF Event 48 Listen Exit” on page 26-47 for information on ENF 48 events.

Appendix A. Using the Unit Verification Service

The information in this appendix describes using the unit verification service to obtain information from the eligible device table. IBM recommends that you use the EDTINFO macro instead; EDTINFO provides more services and is easier to use than the unit verification service.

EDTINFO **must** be used to obtain information on units that are defined as:

- Dynamic,
- Have 4-digit device addresses, or
- Are described by unit control blocks (UCBs) that reside above the 16-megabyte line.

The IEFEB4UV routine interface maybe used, only, to obtain information on units that are static, have 3-digit device addresses and are described as UCBs residing below the 16-megabyte line.

Functions of Unit Verification

The unit verification service (IEFEB4UV routine) enables you to obtain information from the eligible device table (EDT) and to check your device specification against the information in the EDT. See *OS/390 HCD Planning* for information on the EDT.

The unit verification service performs the following functions:

- Check groups
- Check units
- Return unit name
- Return unit control block (UCB) addresses
- Return group ID
- Indicate unit name is a look-up value
- Return look-up value
- Convert device type to look-up value
- Return attributes
- Specify subpool for returned storage
- Return unit names for a device class

Check Groups - Function Code 0

This function determines whether the input device numbers make a valid allocation group. To be valid, the device grouping must include either all the device numbers being verified, or none of them. If this is not the case, the allocation group is split, and the input device numbers do not make up a valid allocation group.

Check Units - Function Code 1

This function determines whether the input device numbers correspond to the unit name in the EDT. In addition to a return code in register 15, it sets to one the high-order flag bit of any device numbers in the parameter list that are not valid.

© Copyright IBM Corp. 1988, 1999

Return Unit Name - Function Code 2

This function returns the unit name associated with a look-up value provided as input. The unit name is the EBCDIC representation of the IBM generic device type (for example, 3390) or the esoteric group name (for example, TAPE) from the EDT.

A look-up value is an internal representation of the unit name, used as an index into the EDT. Because teleprocessing devices do not have generic device names, you cannot use this function to request information about teleprocessing devices.

Note: Do not use this function to determine whether a returned unit name is a generic CTC device or an esoteric group name that contains CTC devices. Instead, use the return attributes function (function code 8) for this purpose.

Return Unit Control Block (UCB) Addresses - Function Code 3

This function returns the UCB pointer list associated with the unit name provided as input.

Return Group ID - Function Code 4

This function returns the allocation group ID corresponding to each UCB address specified in the input list.

Indicate Unit Name is a Look-up Value - Function Code 5

The input to the check units and return UCB addresses functions can be specified as a four-byte internal representation of the unit name rather than as the unit name itself.

Return Look-up Value - Function Code 6

This function returns the four-byte internal representation of the unit name that serves as an index into the EDT. It is the converse of the return unit name function.

Convert Device Type to Look-up Value - Function Code 7

This function will convert a four-byte UCB device type to an internal representation of the unit name, to serve as an index into the EDT. The convert device type to look-up value function allows programs that have only a four-byte UCB device type to query the EDT. It may be used whenever a look-up value is required as input to the unit verification service.

Return Attributes - Function Code 8

This function returns general information about the specified unit name.

Specify Subpool for Returned Storage - Function Code 10

This function is used with the return UCB addresses function or with the return unit names for a device class function. It allows you to specify a particular subpool to return the requested information in.

Return Unit Names for a Device Class - Function Code 11

This function returns a list of IBM generic device types (for example, 3390) and/or esoteric group names (for example, TAPE) associated with the input device class.

Callers of IEFEB4UV

The unit verification routine, IEFEB4UV, is for both problem program callers and for authorized callers. It runs in task mode in the caller's key.

To use IEFEB4UV, the calling program must do the following:

- Create the input data structures and parameter list
- Place the address of an 18-word save area in register 13
- Provide a recovery environment
- Pass control to IEFEB4UV using the LINK and LINKX macro.

On return, IEFEB4UV restores all registers except register 15, which contains a return code.

Input to and Output from Unit Verification Service Routines

You must supply a two-word parameter list when invoking the unit verification routine (IEFEB4UV).

The first word contains the address of a unit table. The contents vary according to the function(s) requested.

The second word contains the address of a 2 byte field (FLAGS), in which you specify the function(s) requested.

The bits in the FLAGS parameter field have the following meanings:

Bit	Function Requested
0	Check groups
1	Check units
2	Return unit name
3	Return UCB addresses
4	Return group ID
5	Indicate unit name is a look-up value
6	Return look-up value
7	Convert device name to a look-up value
8	Return attributes
10	Specify subpool for returned storage
11	Return unit names for a device class
12-15	Reserved for IBM use

Input Parameter List

Figure A-1 on page A-4 shows the input parameter list needed to invoke the unit verification service routine.

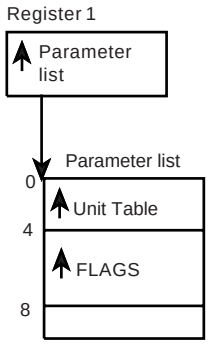


Figure A-1. Input Parameter List

Input and Output Data Structures

The diagrams on the following pages show the input data structures and parameters needed to invoke the unit verification service routine. The output data structure returned by the routine is also shown.

You must declare the structures exactly as shown to get the response indicated by the function(s) you request in FLAGS.

Because many of the input and output data structures are the same, you can request many of the functions in combinations with other functions. The following table lists the valid single functions and combinations of functions that you can request in a single invocation of the unit verification service.

Code
0
0,1
0,1,5
1
1,5
2
2,7
2,8
2,7,8
3
3,5
3,8
3,10
3,5,7
3,5,10
3,8,10
3,5,7,10
4
6
6,8
7
8
10,11
11

Register 15 if Request Fails

On return, register 15 will contain a return code. If the invocation fails, it may be for one of the following reasons:

1. If you request a function that is not valid or a combination of functions that are not valid, register 15 contains a return code of 28 and the request fails.
2. If the JES control table (JESCT) does not contain valid pointers, the environment is incorrect. Register 15 contains a return code of 24. The request fails.

Requesting Function Code 0 (Check Groups)

Input: Set bit 0 in FLAGS to 1.

The input unit table structure is shown below.

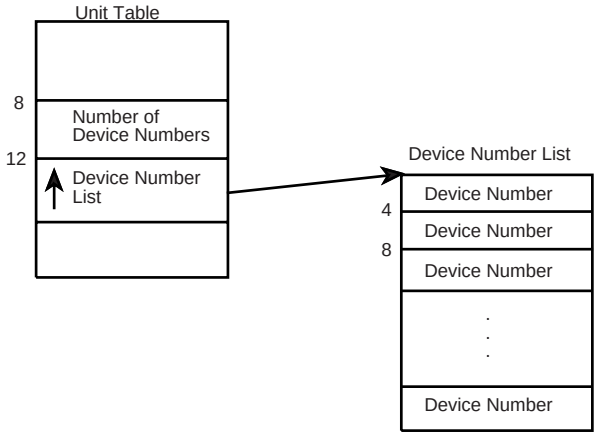


Figure A-2. Requesting Function Code 0 (Check Groups)

Output: None.

Register 15 contains one of the following return codes:

Code	Meaning
0	The specified input is correct.
12	The device groupings are not valid.
28	The required input is not specified or is not valid.

Requesting Function Code 1 (Check Units)

Input: Set bit 1 in FLAGS to 1.

The input unit table structure is shown below.

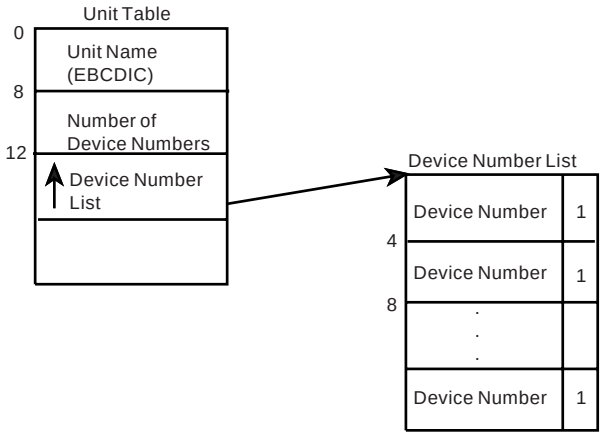


Figure A-3. Requesting Function Code 1 (Check Units)

Output: If a device number is not valid, bit 0 of the FLAG byte is set to 1.

Register 15 contains one of the following return codes:

Code	Meaning
0	The specified input is correct.
4	The specified unit name is not valid.
8	Unit name has incorrect units assigned.
20	One or more device numbers are not valid.
28	The required input is not specified or is not valid.

Requesting Function Code 2 (Return Unit Name)

Input: Set bit 2 in FLAGS to 1.

The input unit table structure is shown below.

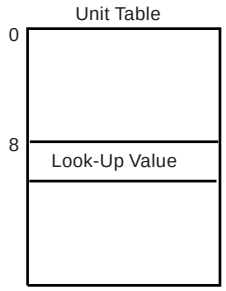


Figure A-4. Requesting Function Code 2 (Return Unit Name)

Output: The unit table contains the unit name as shown in the following figure.

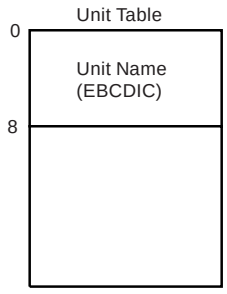


Figure A-5. Output from Function Code 2 (Return Unit Name)

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit table contains the EBCDIC unit name.
4	The look-up value could not be found in the EDT.
28	The required input is not specified or is not valid.

Requesting Function Code 3 (Return UCB Addresses)

Input: Set bit 3 in FLAGS to 1.

The input unit table structure is shown below.

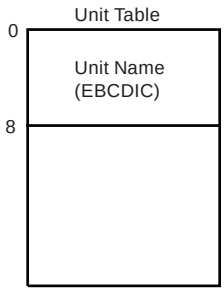


Figure A-6. Requesting Function Code 3 (Return UCB Addresses)

Output: The unit table contains a pointer to the UCB Pointer List as shown in the following figure.

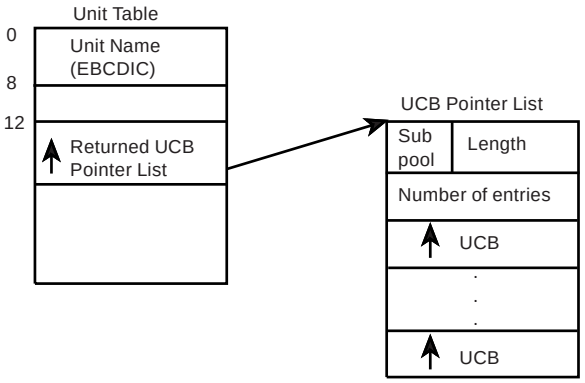


Figure A-7. Output from Function Code 3 (Return UCB Addresses)

For unauthorized callers, the subpool default is 0. See function code 10 for a description of how to change the default subpool. The caller must free the number of bytes in the length field from the subpool before exiting.

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit table contains the pointer to the UCB pointer list.
4	The unit name could not be found in the EDT.
16	Storage was not available for the UCB pointer list.
28	The required input is not specified or is not valid.

Requesting Function Code 4 (Return Group ID)

Input: Set bit 4 in FLAGS to 1.

The input unit table structure is shown below.

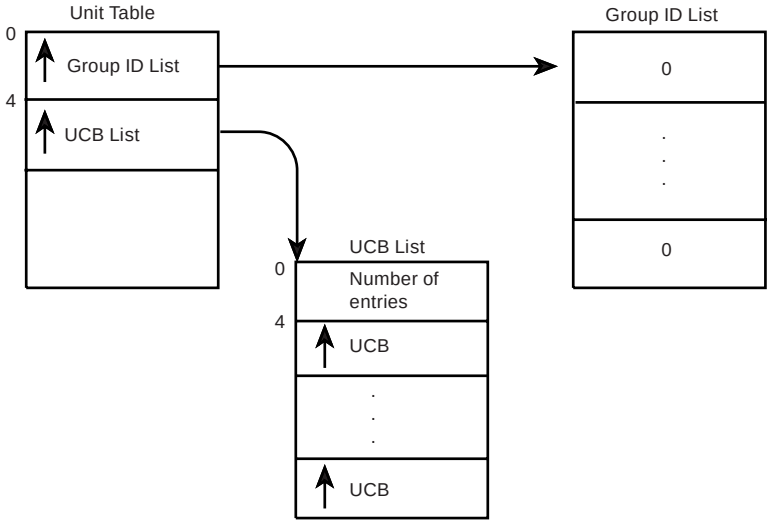


Figure A-8. Requesting Function Code 4 (Return Group ID)

Note: One fullword is provided in the group id list for each UCB in the UCB list. Initialize all entries to zero.

Output: The group id list contains the group id corresponding to each UCB in the input UCB list.

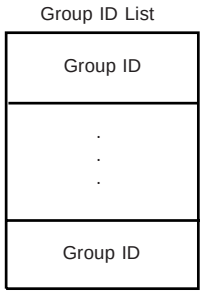


Figure A-9. Output from Function Code 4 (Return Group ID)

Note: If the UCB is not in the EDT, the group id for that particular entry remains zero.

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
28	The required input is not specified or is not valid.

Requesting Function Code 5 (Indicate Unit Name is a Look-up Value)

Input: Set bit 5 in FLAGS to 1.

The input unit table structure is shown below.

This function is not valid by itself. It must be used in combination with other functions that require a unit name as input. If you know the look-up value corresponding to the unit name, you can substitute it for the unit name in the input unit table. The following figure represents the first two fullwords of the unit table when function code 5 is requested.

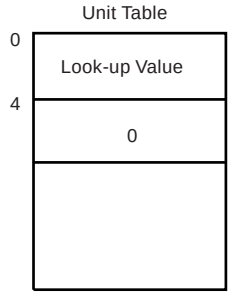


Figure A-10. Requesting Function Code 5 (Indicate Unit Name is a Look-up Value)

Output: None specifically associated with this function.

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
4	The input look-up value could not be found in the EDT.
28	The required input is not specified or is not valid.

Requesting Function Code 6 (Return Look-up Value)

Input: Set bit 6 in FLAGS to 1.

The input unit table structure is shown below.

This function is the opposite of the return unit name function (Code 2). The following figure represents the unit table structure when you request function code 6.

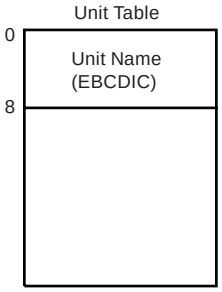


Figure A-11. Requesting Function Code 6 (Return Look-up Value)

Output: The unit table contains the look-up value.

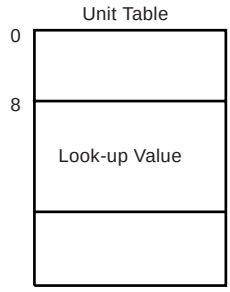


Figure A-12. Output from Function Code 6 (Return Look-up Value)

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
4	The unit name could not be found; no look-up value is returned.
28	The required input is not specified or is not valid.

Requesting Function Code 7 (Convert Device Type to Look-up Value)

Input: Set bit 7 in FLAGS to 1.

The input unit table structure is shown below.

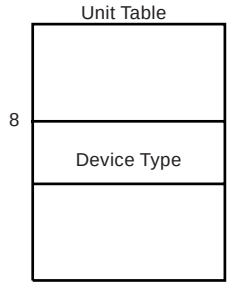


Figure A-13. Requesting Function Code 7 (Convert Device Type to Look-up Value)

Note: The device type is in the format of the UCBTYP field of the UCB.

Output: The unit table contains the look-up value.

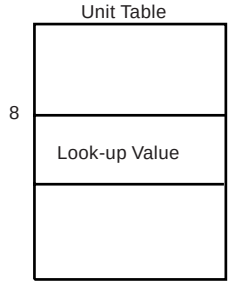


Figure A-14. Output from Function Code 7 (Convert Device Type to Look-up Value)

The conversion of the device type to a look-up value is done in place. There is no error checking of the device type.

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.

4 The input device type is not valid; no look-up value is returned.
28 The required input is not specified or is not valid.

Requesting Function Code 8 (Return Attributes)

Input: Set bit 8 in FLAGS to 1.

The input unit table structure is shown below.

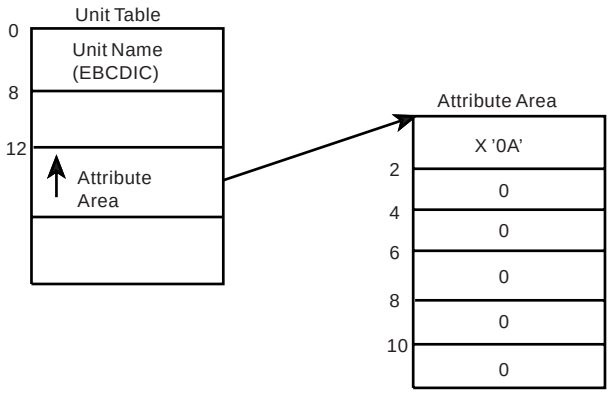


Figure A-15. Requesting Function Code 8 (Return Attributes)

Output: The attribute area contains the following:

Byte	Contents
0	Length of the attribute area (X'0A') This must be filled in prior to calling the unit verification service.
1-2	Flags describing the unit name: • Bit 0 on — unit name is an esoteric group name • Bit 1 on — unit name is VIO-eligible • Bit 2 on — unit name contains 3330V units • Bit 3 on — unit name contains TP class devices • Bits 4-7 are not used.
3	Number of device classes in the unit name
4-7	Number of generic device types in the unit name
8-9	Reserved

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit name was found; the attributes are returned.
4	The unit name was not found; no attributes are returned.
28	The required input is not specified or is not valid.

Requesting Function Code 10 (Specify Subpool for Returned Storage)

Input: Set bit 10 in FLAGS to 1. This function is not valid alone and must be used with either the return UCB addresses function (code 3) or the return unit name function for a device class (code 11). The input unit table structure is shown in the following figure.

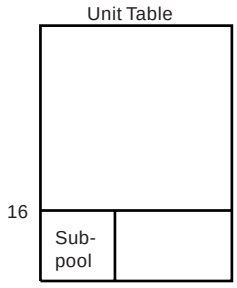


Figure A-16. Requesting Function Code 10 (Specify Subpool for Returned Storage)

Output: See the output from the function that this is invoked in combination with.

The subpool field of the returned list contains the input subpool, and the returned list resides in that subpool. No error checking of the subpool is performed. If the subpool is not valid, the unit verification routine fails.

Requesting Function Code 11 (Return Unit Names for a Device Class)

Input: Set bit 11 in FLAGS to 1.

The following figure shows the input unit table structure.

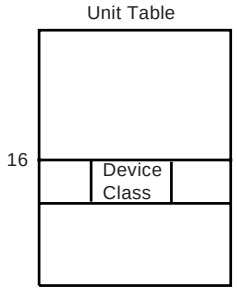


Figure A-17. Requesting Function Code 11 (Return Unit Names for a Device Class)

Output: The unit table contains the pointer to the names list as shown in the following figure.

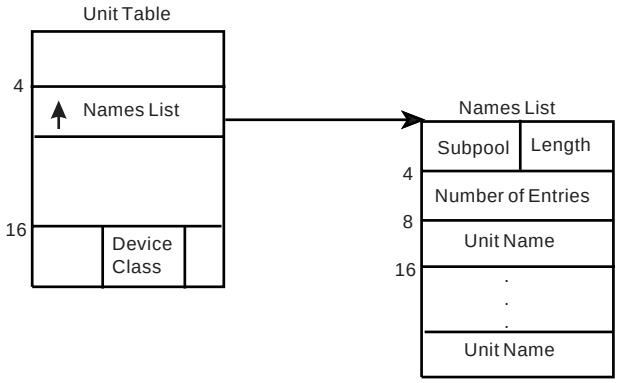


Figure A-18. Output from Function Code 11 (Return Unit Names for a Device Class)

For unauthorized callers, the default subpool is 0. To change this default, see the description for function code 10 (specify subpool for returned storage). The caller must free the number of bytes in the length field from the subpool before exiting.

Register 15 contains one of the following return codes:

Code	Meaning
0	The pointer to the names list is stored in the unit table.
16	Storage was not available for the names list.
28	The required input is not specified or is not valid.

Requesting Multiple Functions - Examples

The following examples show the input to and output from multiple functions.

Example 1 shows the multiple functions of codes 0 and 1.

Example 2 shows the multiple functions of codes 3 and 10.

Example 3 shows the multiple functions of codes 1 and 5.

Example 1 - Function Codes 0 and 1

Input

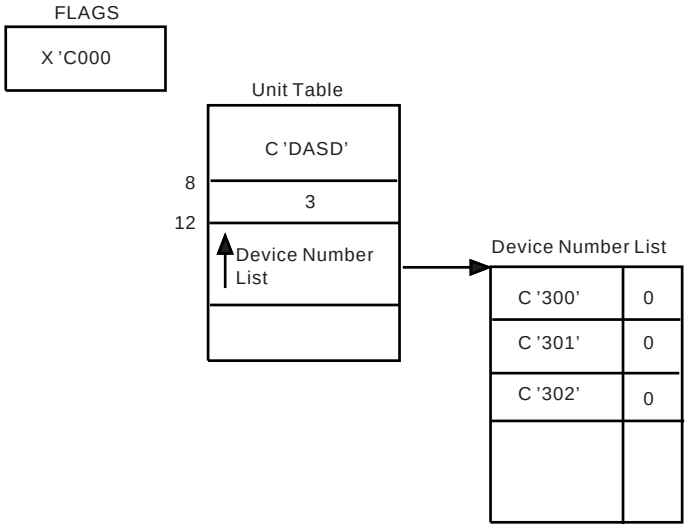


Figure A-19. Input for Function Codes 0 and 1

Output

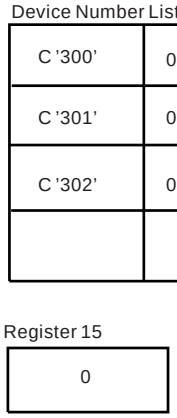


Figure A-20. Output from Function Codes 0 and 1

Note: All input device numbers make up a single allocation group and are associated with the esoteric unit name DASD.

Example 2 - Function Codes 3 and 10

Input

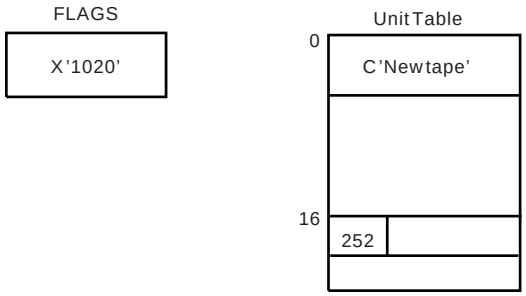


Figure A-21. Input for Function Codes 3 and 10

Output

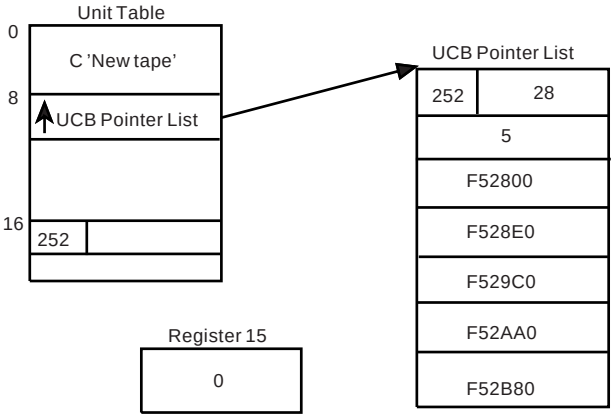


Figure A-22. Output from Function Codes 3 and 10

The caller must free the UCB pointer list before exiting.

Example 3 - Function Codes 1 and 5

Input

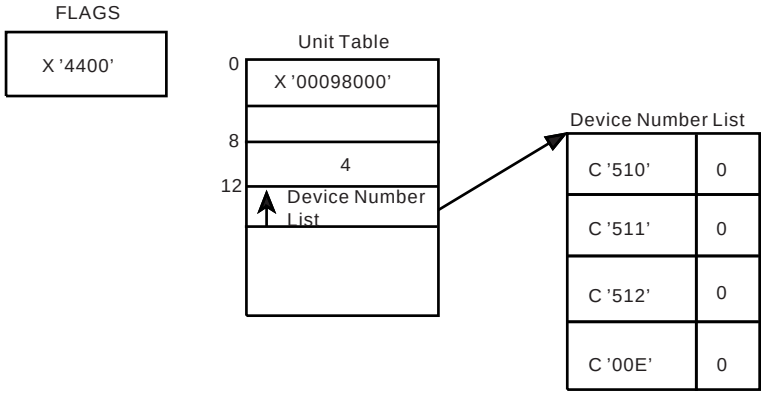


Figure A-23. Input for Function Codes 1 and 5

Output

Device Number List	
C ’510’	0
C ’511’	0
C ’512’	0
C ’00E’	X ’80’

Figure A-24. Output from Function Codes 1 and 5

Note: Device 00E did not belong to the unit name that was associated with the input look-up value.

Appendix B. Using the Virtual Fetch Service

The information in this appendix describes using the virtual fetch service to reduce the time required to locate a load module and bring it into storage for execution. IBM recommends that you use the library lookaside (LLA) with the virtual lookaside facility (VLF). These services also provide improved performance and response time for many applications. LLA is described in *OS/390 MVS Initialization and Tuning Guide* and VLF is described in this book.

The virtual fetch service runs as a started task in its own address space. When virtual fetch manages a module, it places the module in a VIO data set.

Moving load modules from a DASD data set to a VIO data set can reduce contention for channels in your installation. At the same time, the VIO data set is quickly accessible when needed, and virtual fetch relocates the load module's address constants efficiently.

Thus, you can use virtual fetch to improve the responsiveness of interactive subsystems or other large-scale processing programs. Using the virtual fetch service on a system with expanded storage will usually result in greater responsiveness for those load modules in the virtual fetch data set that are frequently referenced.

Virtual fetch can handle any load module that is executable and not in overlay format. However, a load module that virtual fetch is to manage must also be reentrant (read-only). Because virtual fetch runs as a started task, you must define these modules to the service by placing DD statements in the JCL procedure that starts the virtual fetch address space. The DD statements identify the load libraries that virtual fetch is to manage.

You place the virtual fetch JCL procedure in SYS1.PROCLIB and define this procedure. You can further control access to the virtual fetch service by placing the virtual fetch initialization program in a RACF- or password-protected data set.

Functions of Virtual Fetch

Once the service has been initialized, any user can obtain modules from virtual fetch, regardless of the caller's state or key. See *OS/390 MVS Programming: Authorized Assembler Services Guide* for information on initializing the virtual fetch service.

The virtual fetch functions — build, find, and get — are described in the following topics.

After the virtual fetch service has been initialized, a problem program can call virtual fetch to perform three functions:

- To manage a module for the caller (a BUILD request).

  The virtual fetch build function identifies the module that the caller wants to access via the virtual fetch find and get functions.

© Copyright IBM Corp. 1988, 1999

- To determine if virtual fetch is managing a module for the caller (a FIND request). The virtual fetch find function locates the named module and makes it accessible to the caller's address space.
- To pass control to a module (a GET request).

  The virtual fetch get function obtains a copy of the named module and passes control to the named module.

Considerations

To use the virtual fetch service, the calling program must have only a single jobstep TCB in its address space and must meet the following environment requirements:

- Be in TCB mode
- Not be in cross-memory mode
- Not have any outstanding FRRs
- Not hold any locks

The virtual fetch routines that CVTVFIND and CVTVFGET point to cannot be called by an ESTAE-type recovery routine. (In the CVT, the CVTVFIND field points to the build and find routine, and CVTVFGET points to the get routine.)

In addition:

- The FIND and GET functions cannot be used for modules that reside in a partitioned data set extended (PDSE) library.
- Modules managed by virtual fetch cannot use the checkpoint/restart facility. Also, job steps using virtual fetch services cannot use the checkpoint/restart facility.
- IDENTIFY cannot be used against modules obtained by means of a virtual fetch GET request.
- Modules that receive control by means of a GET request are not reused: a fresh copy is obtained for each GET request; only one virtual fetch copy can exist in an address space, and only one caller at a time can use the virtual fetch copy.
- When the build, find, and get functions are invoked, register 1 points to the address of the parameter list (VFPM), which is mapped by macro IHAVFPM (data area VFPM). The format of the parameter list is shown in Figure B-1.

Figure B-1. Virtual Fetch Parameter List

Length	Name	Description
72	VFPMSAVE	18-word register save area that the caller wants passed to the requested program (module).
4	VFPMREG1	A value that the caller wants virtual fetch to pass to the requested program in register 1.
8	VFPMNAME	Name of the requested program (module to be loaded), left-justified with trailing blanks.
1	VFPMLVL	Release level of parameter list. The caller must set this field to the current parameter list level (presently, 0).
1	VFPMFUNC	Depending upon the function you want to perform, set VFPMFUNC to one of the following values: Decimal 1 = BUILD function (VFPMBLD) Decimal 2 = FIND function (VFPMFIND) Decimal 3 = GET function (VFPMGET)
1	VFPMFLAG	Flag byte. **Bit** **Name** **Meaning When Set** 0 VFPMGETM On - Fresh module storage is obtained through GETMAIN and released through FREEMAIN Off - GETMAIN is performed for module storage on the first invocation of the module only. Virtual fetch performs additional GETMAINs only when a refreshed module requires additional storage. Storage is released by means of a page release. 1-7 Reserved.
1	VFPMRTN	Return indicator flag byte set by the GET function. If VFPMRTN is zero, the module specified in VFPMNAME has executed and register 15 contains its return code. If VFPMRTN is not zero,a single bit will be set, as follows: **Bit** **Name** **Meaning When Set** 0 VFPMBUSY The module is in use. The module does not receive control. The caller should retry by invoking the find function. 1 VFPMRESH GET is not able to obtain the requested module. The module does not receive control. The caller should retry by invoking the find function. 2 VFPMAPF An authorized user tried to invoke a module that originally came from a non-APF-authorized library. The module does not receive control. The caller should obtain the module from an authorized library. 3 VFPMBADP Virtual fetch detected an invalid parameter list. Virtual fetch attempts no further processing. The caller should examine the VFPM for incorrect values and retry by invoking the get function. 4 VFPMBADE Virtual fetch encountered an environmental error. (GETMAIN failed, ESTAE failed, etc.). The module does not receive control. Perform clean-up and retry by invoking the find function. 5 VFPMAPPL The requested program abnormally ended. This bit should be examined whenever the caller's recovery routine gains control. Retry by invoking the find function. 6-7 Reserved.

Notes:

1. The caller must set to zero all reserved bits in fields VFPMFLAG and VFPMRTN. Virtual fetch uses the flag bits in VFPMRTN to indicate why a module did not receive control, or that it did receive control but abnormally ended. Only one flag bit is on at a time.
2. The VFPMFUNC field of the parameter list communicates to virtual fetch the function (build, find, or get) that virtual fetch is to perform. You can request only one function per invocation of virtual fetch. After issuing BUILD requests for the required modules, a program must invoke virtual fetch twice more (find and get functions) each time it uses any of the modules.

Programming Conventions for Using Virtual Fetch

- The find and get functions may be invoked in any key, state, or addressing mode; however, virtual fetch assumes that the PSW key matches the caller's TCB key and the jobstep TCB key.
- A caller of virtual fetch must provide a standard 18-word save area, pointed to by register 13; virtual fetch will save the caller's registers in this area. The virtual fetch parameter list contains another 18-word save area, VFPMSAVE, where the called module can save its caller's registers on entry.
- A program must ensure the following register contents when it calls virtual fetch to request that it:
  - Manage a module (a BUILD request)
  - Find a module (a FIND request)
  - Pass control to a program (a GET request).

  Register Contents

  1 Address of the parameter list (VFPM). Any unused high-order address bits of register 1 must be zero.

  13 Address of a standard 18-word save area. Any unused high-order address bits of register 13 must be zero.

  14 Return address. Any unused high-order address bits of register 14 must be zero.

  15 Entry point address of the exit routine.

Requesting Dumps When Using Virtual Fetch

To get a dump of the in-use modules managed by virtual fetch, specify the JPA (job pack area) option either on the SNAP or SNAPX macro or in the appropriate parmlib member.

Note: You may request a certain module from virtual fetch and, concurrently, invoke the same module via a conventional contents supervision mechanism (LINK or LINKX macro, for example). If you then request a dump for that module, both copies will be dumped.

Return Codes from the BUILD Request for Virtual Fetch

When virtual fetch returns control, register 15 contains a return code. The return codes and their meanings are:

Code	Meaning
0	Virtual fetch is able to manage the module.
8	Virtual fetch service is not available.
16	Virtual fetch detected an input parameter error.
20	Virtual fetch detected an environmental error.

Note: A subsystem could invoke virtual fetch's build function during its own initialization processing. The build function does not depend on the prior initialization of the virtual fetch service address space.

Return Codes from the FIND Request for Virtual Fetch

When virtual fetch returns control, register 15 contains a return code. The return codes and their meanings are:

Code	Meaning
0	Virtual fetch found the requested module.
4	Either the module is not managed by virtual fetch or the virtual fetch service's copy of the module is not usable.
8	Virtual fetch service is not available.
12	The requested module previously managed by virtual fetch is not found.
16	Virtual fetch detected an input parameter error.
20	Virtual fetch encountered an environmental error.

Requesting the virtual fetch find function causes virtual fetch to locate and validate the local virtual fetch work area (VFWK) for the requested module. If necessary, virtual fetch copies the VFDE from the current hash table (in the virtual fetch service address space) into the VFWK.

Return Codes from the GET Request for Virtual Fetch

When virtual fetch returns control to its caller, the caller should examine field VFPMRTN in the VFPM. If field VFPMRTN is zero, then the module specified in VFPMNAME received control, and register 15 contains the return code from the specified module. If the module received control, it did so in the state and key, and with the key mask, of the caller of virtual fetch.

If VFPMRTN is not zero, either the named module did not receive control or it ended abnormally. Examine field VFPMRTN to determine the nature of the error.

Note: Modules that are managed by virtual fetch within the caller's address space and are in use when a GET request is made are considered busy (VFPMBUSY is ON) and are not given control. This processing applies to all modules, regardless of their attributes.

Users of virtual fetch can turn off bit VFPMGETM in the parameter list to keep virtual fetch from issuing GETMAINs and FREEMAINs for module storage. (See VFPMFLAG in Figure B-1 on page B-2.) In order to benefit from the performance

gain thus provided, the caller of virtual fetch must be able to tolerate the fact that virtual storage in the caller's address space, for the modules virtual fetch manages, is not freed by means of the FREEMAIN macro.

Figure B-2 is a general illustration of the use of virtual fetch.

```
Let VFPMPTR be the name of register 1, which points to the start of the VFPM area.
      .
      .
      .
   XC  VFPM(VFPMLEN,VFPMPTR),VFPM(VFPMPTR)   *Clear the VFPM area

   MVI VFPMFUNC(VFPMPTR),VFPMBLD             *Indicate BUILD request.
                                              (For a FIND request use
                                              VFPMFIND. For a GET
                                              request use VFPMGET.)

   MVC VFPMNAME(8,VFPMPTR),MODNAME           *Move the name of the
                                              desired module into
                                              the VFPM.

   L   R15,CVTVFIND                          *Set up entry point
                                              address. (For a GET
                                              request use CVTVFGET.)

   BALR R14,R15                              *Invoke virtual fetch
                                              function.

   LTR R15,R15                               *Examine return code.
                                             (For a GET request you
                                             first examine the
                                             VFPMRTN field of the
                                             VFPM.)

   BZ  OK
   .
   .  (Process any errors.)
   .
  OK  (Normal processing continues)
   .
   .
   .
```

```
  MODNAME DC  CL8                           *Name of requested module
  VFPMPTR EQU 1
```

Figure B-2. A Program Using Virtual Fetch

Appendix C. Notices

This information was developed for products and services offered in the USA.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

© Copyright IBM Corp. 1988, 1999

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This book is intended to help the customer to code macros that are available to all assembler language programs. This book documents intended Programming Interfaces that allow the customer to write programs to obtain services of OS/390.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- AFP
- DFSMS/MVS
- DFSMSdfp
- Enterprise Systems Architecture/390
- Hiperspace
- IBM
- IBMLink
- Language Environment
- MVS/DFP
- MVS/ESA
- MVS/SP
- MVS/XA
- OS/390
- RACF
- RMF
- SP
- SP1
- SP2

- Sysplex Timer
- System/370
- S/390

Index

Special Characters

//JOBLIB DD statement 4-16
//STEPLIB DD statement 4-16
/*DEL statement 24-3
/*EOF statement 24-3
/*PURGE statement 24-4
/*SCAN statement 24-4

Numerics

24-bit addressing mode
  description 4-1
  SPIE routine consideration 7-1
31-bit addressing mode
  description 4-1
  SPIE consideration 7-1
46D system completion code 7-1

A

ABDUMP symptom area 9-6
ABEND dump
  requesting 9-2
ABEND macro
  choosing to issue 8-5
  invoking RTM 8-45
  STEP parameter 8-29
abnormal condition
  detect 7-1
  process 7-1
abnormal termination
  ways to avoid with ENQ/DEQ 6-14
  when deleting a SPIE/ESPIE environment 7-1
  when issuing CLOSE 11-14
access list
  adding an entry 14-9
  adding entry for data space 14-9
  adding entry for hiperspace 15-31
  definition 14-2
  type 14-3
access list entry
  adding 14-9
  deleting 14-10
access register
  use 14-2, 15-7
  using 14-1
access to a data object
  permanent object 16-6
  temporary object 16-7
add an entry to an access list
  description 14-9
  example 14-9
address space control mode
  definition 14-1
  switching 14-1
address space priority 3-2
addressing mode
  affect on BAL and BALR instruction 4-2
  bit in the PSW 4-2
  changing
    example 4-4
    using BSM or BASSM 4-3
  considerations when passing control 4-3
  indicator
    in a PDS entry 4-1
    in an entry point address 4-4, 4-16
  of alias entry point 4-29
  of SPIE routine 7-1
  specifying
    in source code 4-1
    using linkage editor control card 4-1
ALESERV macro
  ADD request
    example 14-9, 15-31
    use 14-9, 15-31
  DELETE request
    example 14-10
  example 15-20
ALET
  ALET qualified address 2-1
  purpose 2-1
  when ALET qualification is required 2-1
ALET (access list entry token)
  definition 14-2
  example of loading a zero into an AR 14-8
  loading into an AR 14-8
  use 14-2
ALET (address list entry token)
  for primary address space 14-6
  with a value of zero 14-6
ALET-qualified address
  used in macro parameter list 14-10
algorithm
  run length encoding 22-1
  used by data compression service 22-1
  used by data expansion service 22-1
alias
  addressing mode 4-29
  establishing 4-29
AMODE program attribute
  changing
    example 4-3
    using BSM or BASSM 4-3
  indicator
    in a PDS entry 4-1

© Copyright IBM Corp. 1988, 1999

AMODE program attribute *(continued)*
indicator *(continued)*
in an entry point address 4-4, 4-16
purpose 4-1
specifying
in source code 4-1
using linkage editor control card 4-1
value 4-2
anchor 12-1
answer area
system logger
size 26-18
APF-authorization
when needed by problem state program 4-16
application resource
releasing through recovery 8-3
AR (access register)
example of loading an ALET of zero into 14-8
rules for coding 14-6
use 14-2, 15-7
using 14-1
AR information
formatting and displaying 14-11
AR instruction
for manipulating contents of ARs 14-7
AR mode
coding instructions 14-6
description 14-1
example 14-5
importance of comma 14-7
importance of the contents of ARs 14-8
issuing macros 14-10
rules for coding 14-6
use 14-2
writing programs 14-5
AR mode program
call a primary mode program 2-9
call an AR mode program 2-9
defined 2-1
linkage procedure 2-8
pass parameters 2-10
receive control from a caller 2-8
return control to a caller 2-9
ARCHECK subcommand
formatting and displaying AR information 14-11
architecture level 10-1
ARR (associated recovery routine)
choosing 8-8
ASC mode
AR mode program defined 2-1
definition 14-1
primary mode program, defined 2-1
switching 14-1
switching mode 2-2
when control is received 2-2
assembler example
window service 16-20
assembler instruction
examples of use in AR mode 14-8
use in AR mode 14-5, 14-6, 14-8
ATTACH and ATTACHX macros
ASYNCH parameter 8-29
defining a recovery routine 8-8
ECB parameter 8-46
ESTAI parameter 8-8, 8-20
ETXR parameter 8-46
PURGE parameter 8-28
STAI parameter 8-8
TERM parameter 8-30
ATTACH macro
addressing mode consideration 4-16
creating subpools 11-9
DPMOD parameter 3-2
ECB parameter, use 3-4
ETXR parameter, use 3-4
example of sharing DU-ALs 15-20
GSPL parameter 11-9
GSPV parameter 11-9
LPMOD parameter 3-2
requesting subpool ownership 11-9
sharing a DU-AL with subtask 15-20
SHSPL parameter 11-9
SHSPV parameter 11-9
specify subpools 11-9
SZERO parameter 11-9
TASKLIB parameter 4-17, 4-18
use 3-1, 4-7, 4-16
authorization requirements
IXGCONN service 26-24
system logger 26-11
system logger application program 26-11
system logger couple data set 26-11
system logger, IXCMIAPU utility 26-11
availability
increasing through recovery 8-3

B

BAL instruction 4-2
BALR instruction 4-2
BAS (branch and save) instruction 4-2
BAS instruction 4-2
base register
establishing 2-6
BASR (branch and save instruction register form) 4-2
BASR instruction 4-2
BASSM (branch and save and set mode) 4-3
BASSM instruction 4-3, 4-25
BLDL macro 4-20, 4-24, 4-25

BLDMPB macro 21-17
blocks of an object
definition 16-1
identifying blocks to be viewed 16-13
size 16-1
updating blocks in a temporary object 16-16
updating blocks on DASD 16-17
BLOCKS parameter on DSPSERV 15-9, 15-16, 15-29, 15-34
branch instruction
BAL (branch and link) 4-2
BALR (branch and link register) 4-2
BAS instruction 4-2
BASR instruction 4-2
BASSM instruction 4-3
BSM instruction 4-3
use 4-25
use with XCTL, danger 4-27
branching table
use in analyzing return codes 4-12
bring a load module into virtual storage 4-16
BSM (branch and set mode) 4-3
BSM instruction 4-3
build a symptom record 10-3
BUILD request
for virtual fetch B-5

C

CALL macro
use 4-10, 4-11, 4-22, 4-25
callable cell pool service
advantages of using 12-1
compared to the CPOOL macro 12-1
data space 15-17
data space example 15-18
calling program
definition 2-1
calling sequence identifier 4-30
cell 11-6, 12-2
allocating 12-5
deallocating 12-5
cell pool 12-2
activating 12-4
anchor 12-1
contracting 12-4
creating 12-4
deactivating 12-4
disconnecting 12-4
expanding 12-4
extent 12-1
obtaining 11-6
obtaining status about 12-5
size 12-2
storage 12-2
cell pool service
CSRPACT service 12-5
CSRPBLD service 12-4
CSRPCON service 12-5
CSRPDAC service 12-4
CSRPDIS service 12-4
CSRPEXP service 12-4
CSRPFR1 service 12-5
CSRPFRE service 12-5
CSRPGET service 12-5
CSRPGT1 service 12-5
CSRPRFR service 12-5
CSRPRFR1 service 12-5
CSRPRGT service 12-5
CSRPRGT1 service 12-5
query 12-5
return code 12-6
type of service 12-2
control 12-4
cell storage 12-2
central storage
sharing through IARVSERV macro 19-1
CHAP macro
use 3-3
characters printed on an MCS console 20-5
check groups function of unit verification service A-1
check units function of unit verification service A-1
checkpoint 17-6
checkpoint/restart
manage storage 15-25, 15-41
CHKPT macro 17-6
CHNGDUMP command 9-2
choose the name of a data space 15-9
code
descriptor 20-7
message routing 20-6
code instructions in AR mode 14-6
comma
careful use of in AR mode 14-7
communication
in a recovery environment 8-19
in a sysplex environment 20-10
provided by recovery 8-4
compiler
message 21-9
invoking 21-9
complete exit
system logger
environment 26-14
information passed to 26-13
inputs 26-14
IXGCONN service 26-13
programming 26-14

compress data
steps required 22-2
using the data compression service 22-1
compression
of data
description 22-9
symbols 22-5
concatenated data sets 4-17
concurrent request for resource
limiting 6-12
connect
allocating coupling facility space 26-24
to a log stream 26-22
console
CONVCON macro 20-12
determining status 20-13
determining the name or ID 20-12
parameter list
initializing field 20-12
retrieving information 20-12
validating a name or ID 20-13
validating area ID 20-14
control
return 4-13, 4-15, 4-26
control virtual storage 11-6
CONVCON macro
parameter list 20-12
retrieving console information 20-12
convention
for passing control 4-7
convert device type to look-up value function of unit verification service A-2
converting central to virtual address 19-9
CONVTOD macro
using 20-2
couple data set
LOGR 26-7
system logger 26-7
coupling facility
structure storage limit reached 26-33
system logger
damage 26-55
failure 26-55, 26-57
full condition 26-57
loss of connectivity to 26-56
system logger data on 26-3
coupling facility structure
system logger
storage limit reached 26-33
coupling facility version number
IXGQUERY service 26-44
CPOOL macro
use 11-6
CPUTIMER macro
using 20-2
CPYA instruction
description 14-8
example 14-8
create
hiperspace 15-25
subpool 11-9
task 3-1
create data space
example 14-9, 15-12, 15-23
rules for 15-8
create hiperspace
example 15-28
CSRCESRV macro
using with an MVS/ESA system 22-2
using with an MVS/XA system 22-4
CSRCEXA load module 22-4
CSRCMPSC macro 22-4, 22-9
CSREVW window service
defining a view of an object 16-11
CSRIDAC window service
obtaining access to a data object 16-9
terminating access to an object 16-19
CSRL16J service
passing control with all registers intact 4-4
CSRREFR window service
refreshing changed data 16-16
CSRSAVE window service
updating a permanent object on DASD 16-17
CSRSCOT window service
saving interim changes in a scroll area 16-15
updating a temporary object 16-16
CSRVIEW window service
defining a view of an object 16-11
terminating a view of an object 16-17
CSVINFO macro
Comparison with CSVQUERY 4-30
MIPR 4-30
CSVQUERY macro
Comparison with CSVINFO 4-30
current size of data space 15-10
CVT (communication vector table)
CVTVFIND field for virtual fetch B-2

D

DAE (dump analysis and elimination)
dump not suppressed 9-8
dump suppression 9-2
providing information through recovery 8-15
DASD (direct access storage device)
data transfer from by window services 16-3
updating a permanent object on DASD 16-17
DASD log data set
space 26-58
DASD log data sets
SMS management of 26-8

DASD-only
log stream 26-22
DASD-only log stream
system logger data on 26-4
data compression and expansion service
data which can exploit 22-1
recovery routine 22-2
summary table 22-2
data compression and expansion services
using 22-1
data compression service
steps required to compress data 22-2
using 22-1
data control block
deleting load modules that contain 11-13
SNAP dump 9-9
data expansion service
running under an MVS/ESA system 22-2
running under an MVS/XA system 22-3
using 22-1
data object
defining a view 16-11
defining multiple views 16-14
extending the size 16-14
identifying 16-9
mapping 16-1
multiple objects, example 16-5
multiple windows to an object, example 16-5
scroll area to DASD, example 16-4
temporary object, example 16-4
window to a scroll area, example 16-4
window to DASD, example 16-3
obtaining
access to a data object, overview 16-8
access to a data object, procedure for 16-9
scroll area 16-10
refreshing changed data 16-16
saving interim changes 16-15
specifying type of access 16-10
structure 16-1
terminating access to an object 16-19
updating a temporary object 16-16
data set
dump 9-9
data sharing with IARVSERV macro 19-1
data space
access data 15-7
choosing the name for 15-9
compared to address space 15-1
create 15-7
creating 15-8
default size 15-3
definition 15-1, 15-7
delete 15-7
deleting 15-17
dumping storage 15-25
data space *(continued)*
establish addressability 15-12
example 14-9, 15-2
example of creating 15-23
example of deleting 15-23
example of moving data into and out 15-12
example of using 15-23
extending current size 15-14
identifying the origin 15-11
illustration 15-1
loading pages into central storage 15-16
managing storage 15-3
paging out of central storage 15-16
release storage 15-7
restoring 15-25
rules for using 15-7
saving 15-25
SCOPE=ALL 15-9
SCOPE=COMMON 15-9
SCOPE=SINGLE 15-9
shared between two programs 15-19, 15-20
sharing with IARVSERV macro 19-1
specify size 15-10
storage available for 15-3
use 15-2, 15-7
using efficiently 15-23
data space storage
managing 15-3
release 15-7
releasing 15-16
rules for releasing 15-16
data to be viewed 16-13
data-in-virtual
mapping a hiperspace object to an address space window 15-39
mapping into hiperspace 15-37, 15-38
data-in-virtual object
definition 13-1, 16-1
mapped into data space storage 15-22
data-in-virtual window
requirements for 13-1
date and time of day
obtaining 20-1
DCB parameter 4-19
DD statements required for dumps 9-2
DE parameter 4-20
debugging aid for call sequence 4-30
default priority 3-2
define multiple views of an object 16-14
define the expected window reference pattern 16-12
define view of a data object 16-11
define window disposition 16-12
delete
access list entry 14-10
example 14-10

delete *(continued)*
data space 15-4
example 14-10
hiperspace 15-4
description 15-35
example 15-41
delete data space
example 15-23
delete data spaces
rules 15-17
DELETE macro
lowering the responsibility count 11-13
delete message 20-5
delete messages already written 20-11
deleting
from a log stream 26-38
illustration 26-39
DEQ macro 6-17
rules for using 6-10
use 6-10, 6-14
descriptor code 20-7
DETACH macro
use 3-4
device
verification A-1
device class
unit name A-3
DFP requirement for window service 16-10
dictionary
compression 22-5
entries 22-10
expansion 22-5
directory entry
PDS (partitioned data set) 4-1
disconnecting
from a log stream 26-22
dispatching priority
assign 3-2
display AR information 14-11
DIV (data-in-virtual) service
restrictions
when using IARVSERV macro 19-8, 19-9
DIV macro
example 15-40
example of mapping object into data space 15-22
mapping a data-in-virtual object to a hiperspace
example 15-38
mapping a hiperspace as a data-in-virtual object
example 15-40
mapping object to a data space 15-22
programming example 13-22
retain mode 13-14, 13-18, 13-20
rules for invoking 13-22
sharing data in a window among tasks 13-21
use 13-4, 15-37
save 13-15
unaccess 13-21
DIV macro *(continued)*
use *(continued)*
unidentify 13-21
unmap 13-20
using data-in virtual 13-1
when to use data-in-virtual 13-2
DOM macro
function 20-11
DPMOD parameter on ATTACH 3-2
DPRTY parameter on the EXEC statement 3-2
DSPSERV macro
CREATE request
example 14-9, 15-12, 15-29, 15-35, 15-38, 15-40
DELETE request
example 14-10, 15-35, 15-39, 15-40
use 18-1
EXTEND request
example 15-15
LOAD service
use 15-16
OUT service
use 15-16
RELEASE request
use 15-34, 18-2
rules 15-16
DU-AL
add entry 15-7
compared with a PASN-AL 14-3
definition 14-3
illustration 14-3
dump
ABEND dump 9-1
data sets for 9-9
index in SNAP dump 9-9
not suppressed 9-8
requesting 9-1
requesting in recovery 8-15
select type 9-1
SNAP dump 9-1, 9-9
summary 9-9
suppression 9-2
symptom 9-2
SYSABEND dump 9-1
SYSMDUMP dump 9-1
SYSUDUMP dump 9-1
Transaction dump 9-1, 9-10
types a problem program can request 9-1
dump service 9-1
dump storage in a data space 15-25
duplicate
names in unique task libraries, 4-19
resource request 6-14
dynamic I/O configuration change
detecting 23-1
dynamic load module structure
advantage 4-7

dynamic load module structure *(continued)*
description 4-6, 4-7

E

EAR instruction
description 14-8
ECB (event control block)
description 6-2
parameter of ATTACH 3-4, 3-5, 6-2
EDT (eligible device table)
description 23-4
obtaining information 23-4
unit verification service
IEFEB4UV routine A-1
EDTINFO macro 23-4
end-of-task exit routine 3-5
ENF event code 48
listen exit
writing 26-47
system logger
connect service 26-27
system logger application 26-19
ENQ macro 6-17
example 6-13
rules for using 6-10
use 4-26, 6-10
entry point
adding 4-29
address
AMODE indicator 4-4
alias use 4-29
identifier 4-30
identify 4-10
EP parameter 4-18
EPIE (extended program interruption element) 7-4
EPLOC parameter 4-18
error
recovering from software 8-1
ESD (external symbol dictionary)
AMODE/RMODE indicator 4-1
ESO hiperspace
definition 15-27
ESPIE environment
deleting 7-1
establishing 7-1
ESPIE macro
option
RESET parameter 7-3
SET parameter 7-3
TEST parameter 7-3
use 7-1
using 7-3
establish addressability to a data space 15-12
definition 14-2
example 15-23
ESTAE and ESTAEX macros
0 parameter 8-7
ASYNCH parameter 8-29
CT parameter 8-7, 8-46
defining a recovery routine 8-7
OV parameter 8-46
PARAM parameter 8-20
PURGE parameter 8-28
TERM parameter 8-30
ESTAE and ESTAEX routine
definition 8-8
ESTAE-type recovery routine (extended specify task abnormal exit)
See also recovery
providing 8-7
ESTAI routine
definition 8-8
ETR (External Time Reference hardware facility)
checking for TOD-clock synchronization 20-1
ETXR parameter of ATTACH
use 3-4
event
signal completion 6-2
EVENTS macro
use 6-2
example
data object mapped to a window 16-2
mapping
multiple objects 16-5
object to multiple windows 16-5
permanent object that has a scroll area 16-4
permanent object that has no scroll area 16-3
temporary object 16-4
structure of a data object 16-2
window services coding example 16-20
exclusive resource control 6-12
EXEC statement
DPRTY parameter 3-2
exit routine
altering register content 7-5
altering the old PSW 7-5
end-of-task 3-5
functions performed by 7-5
register contents on entry 7-4
specifying 7-1
using serially reusable resources 6-9
expand data
steps required under an MVS/ESA system 22-2
steps required under an MVS/XA system 22-4
expanded data
using the data expansion service 22-1
expansion
of data
description 22-10
explicit requests for virtual storage 11-1

extend current size of data space
example 15-15
procedure 15-14
extend current size of hiperspace
procedure for 15-34
EXTEND parameter on DSPSERV 15-14, 15-34
extend the size of an object 16-14
extent 12-1

F

find
load module 4-17
FIND request
virtual fetch B-5
format AR information 14-11
frame
assigning 18-1
repossessing 18-1
FREEMAIN macro
use 11-1, 11-5

G

gap in reference pattern service
defining 18-8
definition 18-8
gap in reference pattern services
definition 18-8
GENNAME parameter on DSPSERV 15-8, 15-9, 15-28
GET request
virtual fetch B-5
GETMAIN macro
creating subpools 11-9
LOC parameter 11-2, 11-3
requesting storage at an address 11-3
type 11-2
use 11-2
gigabyte 4-1
global resource 6-11
global resource serialization 6-17
GQSCAN macro
function 6-17
result 6-20
TOKEN parameter 6-17

H

hiperspace
as data-in-virtual object 15-39
compared to address space 15-1
creating 15-25, 15-28
default size 15-3
definition 15-1, 15-25
deleting 15-35
hiperspace *(continued)*
extending current size 15-34
fast data transfer 15-31
illustration 15-1
managing storage 15-3
manipulating data
illustration 15-26
mapping data-in-virtual object into 15-37, 15-38
referencing data 15-29
releasing storage 15-34
restoring 15-41
saving 15-41
shared between two programs 15-31
specify size 15-10
storage available for 15-3
two types 15-26
window services use 16-3
hiperspace storage
managing 15-3
releasing 15-34
rules for releasing 15-34
HSPALET parameter on HSPSERV macro 15-31
HSPSERV macro
example 15-31
read operation 15-29, 15-30, 15-31
SREAD and SWRITE operation
example 15-35
illustration 15-29
use of move-page facility 15-31
write operation 15-29
HSTYPE parameter on DSPSERV 15-28

I

I/O configuration change
detecting 23-1
I/O configuration token
detecting I/O configuration changes with 23-1
IARR2V macro
ASID parameter 19-9
converting a central storage address to virtual 19-9
IARVSERV sharing effectiveness 19-9
NUMVALID parameter 19-9
NUMVIEW parameter 19-9
RSA parameter 19-9
STOKEN parameter 19-9
VSA parameter 19-9
WORKREG parameter 19-9
IARVSERV macro
CHANGEACCESS parameter 19-4
copy-on-write 19-6
CVT mapping macro hardware check 19-7
data sharing 19-1
diagnostics 19-9
example of use 19-7
IARVRL mapping macro
required fields 19-6

IARVSERV macro *(continued)*
parameters description 19-6
RANGLIST parameter 19-6
READONLY parameter 19-6
restrictions using DIV (data-in-virtual) service 19-9
restrictions using DIV (data-in-virtual) services 19-8
RETAIN parameter 19-7
SHARE parameter 19-4
SHAREDWRITE parameter 19-6
sharing effectiveness 19-9
SHRDATA IPCS subcommand 19-9
TARGET_VIEW parameter 19-6
types of views 19-3
UNIQUEWRITE parameter 19-6
identify a data object 16-9
identify a window 16-11
identify blocks to be viewed 16-13
identify the origin of the data space 15-11
IEALSQRY macro
tracking entries in the linkage stack 8-41
IEANTCR callable service 17-1
IEANTDL callable service 17-1
IEANTRT callable service 17-1
IEFEB4UV routine A-1
authorized caller A-3
caller's function A-3
key A-3
mode A-3
problem program caller A-3
IHAVFPM mapping macro B-2
implicit requests for virtual storage 11-11
import connection
IXGCONN service
IMPORTCONNECT parameter 26-26
import log blocks
IXGIMPRT service 26-39, 26-40, 26-41
INADDR parameter on the GETMAIN macro 11-3
INADDR parameter on the STORAGE macro 11-5
initial size of data space 15-10
initiate offload
IXGOFFLD service 26-45
inline parameter list
use 4-10
installation limit
amount of storage for data space and hiperspace 15-10
on amount of storage for data space and hiperspace 15-3
on size of data space and hiperspace 15-3
size of data space 15-10
interlock
avoiding 6-15
illustration 6-16
internal reader facility 24-1
allocating the data set 24-2
through dynamic allocation 24-2
through JCL 24-2
internal reader facility *(continued)*
closing the data set 24-3
coding /*DEL 24-3
coding /*EOF 24-3
coding /*PURGE 24-4
definition 24-1
example 24-4
opening the data set 24-2
sending records to the data set 24-3
setting up and using 24-1
tasks involved in using 24-1
interval timing, establish 20-2
INTRDR data set 24-2
IOCINFO macro 23-1
IPCS (interactive program control system)
formatting and displaying AR information 14-11
issue macros in AR mode 14-10
IXGANSAA macro
answer area mapping 26-18
IXGBRWSE service
browse cursor 26-35
browse session 26-34
browse token 26-34
read data from a log stream 26-34
REQUEST=READBLOCK 26-35
REQUEST=READCURSOR 26-35
REQUEST=RESET 26-35
REQUEST=START 26-35
searching for a log block by time stamp 26-36
illustration 26-37
with IXGDELET service 26-37
with IXGWRITE service 26-37
IXGCONN service
allocating coupling facility space at connection 26-24
authorization requirements 26-24
complete exit 26-13
environment 26-14
information passed to 26-13
inputs 26-14
programming 26-14
connect process and staging data sets 26-24
connecting to and disconnection from a log stream 26-22, 26-23
disconnect from a log stream 26-27
ENF event code 48 and 26-27
import connection 26-26
resource manager
connecting as 26-25
using ENF event 48 with 26-26
resource manager exit 26-28
user data for a log stream
USERDATA parameter 26-27
write connection 26-26
IXGDELET service
and resource manager exit processing 26-39

IXGDELET service *(continued)*
delete data from a log stream
illustration 26-39
deleting data from a log stream 26-38
IXGIMPRT service 26-39
import log blocks 26-40, 26-41
manage a target log stream 26-46
safe import point 26-41
IXGINVNT service
DASDONLY parameter 26-22
managing the LOGR policy 26-20
MODEL parameter 26-21
example 26-21
IXGOFFLD service
initiate offload 26-45
manage a target log stream 26-46
IXGQUERY service
coupling facility version number 26-44
log stream information 26-41
manage a target log stream 26-46
safe import point 26-41
IXGUPDAT service
modify log stream control information 26-46
time stamp 26-46
IXGWRITE service
BUFFALET parameter 26-32
log block buffer 26-32
BUFFKEY parameter 26-32
BUFFLEN parameter 26-32
sequence of log blocks 26-32
write to a log stream 26-32

J

JCL
for LOGR subsystem 26-51
JES (job entry subsystem)
and the internal reader 24-1, 24-2, 24-3
job library
reason for limiting size 4-20
use 4-16
when to define 4-20
job output
sending to the internal reader 24-3
job step task
create 3-1
JPA (job pack area) 4-17

L

LAE instruction
description 14-8
example 14-8
LAM instruction
description 14-8
example 14-8, 14-9
language
checking availability 21-14
library
description 4-17
search 4-17
limit priority 3-2
linear data set
creating a 13-3
link library 4-16
LINK macro
addressing mode consideration 4-16
use 4-16, 4-22, 4-24
when to use 11-13
linkage
consideration 4-2
editor 4-1
linkage convention
advantages of using the linkage stack 2-3
AR mode program linkage procedure 2-8
AR mode program, defined 2-1
establish a base register 2-6
for branch instruction 2-1
introduction 2-1
parameter convention 2-9
primary mode program linkage procedure 2-6
primary mode program, defined 2-1
register save area, provide 2-2
register, saving 2-2
using a caller-provided save area 2-4
using the linkage stack 2-3
linkage stack
advantages of using 2-3
at time of retry 8-41
considerations for ESTAE-type recovery routines 8-27
example of using the 2-3
how to use 2-3
LINKX macro
use 4-22
load
registers and pass control 4-9
virtual storage 18-3
load an ALET into an AR 14-8
load instruction in AR mode
example 14-7
load list area 4-17
LOAD macro
indicating addressing mode 4-16
use 4-16, 4-22, 4-25
when to use 11-13
load module
alias 4-29
characteristic 4-6
execution 4-7
how to avoid getting an unusable copy 4-21
location 4-16

load module *(continued)*
more than one version 4-18
name 4-29
search for 4-17
structure type 4-6
use count 4-23, 4-27
using an existing copy 4-21
load module execution 4-7
loaded module
information 4-30
LOC parameter on the GETMAIN macro 11-2, 11-3
LOC parameter on the STORAGE macro
requesting storage at an address 11-5
local resource 6-11
location of a load module 4-16
log block buffer 26-32
BUFFALET parameter 26-32
BUFFKEY parameter 26-32
BUFFLEN parameter 26-32
log data
on DASD log data sets 26-3, 26-4, 26-8
log data sets 26-3, 26-8
allocation 26-4, 26-8
log stream 26-2
connection to
IXGCONN service 26-22
DASD-only 26-22
definition 26-1
delete data from
illustration 26-39
deleting data from
IXGDELET service 26-38
different ways of connecting to 26-23
disconnection from
IXGCONN service 26-22
gaps in 26-15
illustration 26-1
model 26-21
example 26-21
read from
IXGBRWSE service 26-34
write to
IXGWRITE service 26-32
writing to
sequence of log blocks 26-32
log stream information
IXGQUERY service 26-41
log stream time stamp
IXGUPDAT service 26-46
LOGR
couple data set 26-7
policy 26-7
LOGR couple data set 26-7
LOGR policy 26-7
managing
IXGINVNT service 26-20
LOGR subsystem 26-49
JCL for 26-51
read log data in data set format
eligible applications 26-49
using 26-51
Logrec Data Set
description 10-1
look-up value for the EDT
See also unit verification service
defined A-2
obtaining A-2
LPA (link pack area) 4-17
LPMOD parameter on ATTACH 3-2
LQB (language query block) 21-14

M

macro
form
execute 11-11
list 11-11
standard 11-11
issuing in AR mode 14-10
reenterable form 11-11
way of passing parameters 11-11
mainline routine
definition in a recovery environment 8-6
manage a target log stream 26-46
managing the LOGR policy
IXGINVNT service 26-20
manipulate data in hiperspace 15-26
manipulate the contents of ARs 14-7
map data-in-virtual object into data space
rules for problem state program 15-22
map data-in-virtual object into hiperspace 15-38
example 15-38
rules for problem state program 15-37
map hiperspace as data-in-virtual object 15-39
example 15-40
map object into data space
using DIV macro 15-22
map object to a data space
using DIV macro 15-22
maximum size of data space 15-10
MCS console
characters displayed 20-5
megabyte 4-1
member names
establish 4-29
message
deleting 20-5, 20-11
descriptor code 20-7
disposition 20-7
example of WTO 20-8
identifier 20-9
indicator in first character 20-7

message *(continued)*
MLWTO (multiple-line) 20-6
replying 20-9
routing 20-6, 20-7
single-line 20-6
translating 21-1
writing 20-5
message compiler 21-9
invoking 21-9
message file
compiling 21-9
message skeleton
creating 21-4
format 21-5
validating 21-7
message text
format 21-6
MIPR
CSVINFO macro 4-30
MLWTO (multiple-line) message
considerations for using 20-6
MMS (MVS message service) 21-1, 21-21
coding example 21-19
support for additional language 21-18
mode
primary 14-1
MODE parameter 26-11
MODE=ASNYCNORESPONSE parameter 26-12
MODE=SYNCEXIT parameter 26-12
modify log stream control information
IXGUPDAT service 26-46
move data between hiperspace and address space 15-29
move-page facility 15-31
MPB (message parameter block) 21-17
building 21-17
using BLDMPB and UPDTMPB 21-17
using for new message 21-17
multiple versions of load modules 4-18
MVS macro
issuing in AR mode 14-10

N

name
resource 6-10
name a data space 15-9
NAME parameter on DSPSERV 15-8, 15-9, 15-28
name/token callable service
link-editing with your application 17-6
use of the service 17-1
name/token pair
creating 17-3
deciding which to use 17-3
definition 17-1
deleting 17-3
name/token pair *(continued)*
home 17-5
level 17-5
home address space 17-2
primary address space 17-2
system 17-2
task 17-2, 17-3
retrieving the token 17-3
non-reenterable load module 11-13
non-shared standard hiperspace
creating 15-28
definition 15-28
Notices C-1
NUMRANGE parameter on HSPSERV 15-30

O

obtain access to a data object 16-9
operator
consoles, characters displayed 20-5
messages, writing 20-5
origin of data space 15-11
originating task 3-1
OUTNAME parameter on DSPSERV 15-8, 15-9
overlay load module structure 4-6

P

page
faults, decreasing 18-5
movement 18-1
size 18-1
page out virtual storage 18-3
page-ahead function 18-3
paging I/O 18-1
paging service
input 18-4
list of services 18-1
parallel execution
when to choose 3-1
parameter convention 2-9
parameter list
description 4-8
example of passing 4-9
indicate end 4-11
inline, use 4-10
location 4-27
parameter list for AR mode program
illustration 2-11
PASN-AL
add entry 15-7
compared with a DU-AL 14-3
definition 14-3
pass control
between control sections 4-8
between programs with all registers intact 4-4

pass control *(continued)*
between programs with different AMODEs 4-3, 4-25
between programs with the same AMODE 4-3
in a dynamic structure 4-16, 4-28
with return 4-22
without return 4-26
in a simple structure 4-7, 4-15
with return 4-10
without return 4-8
preparation 4-8
prepare 4-10
using a branch instruction 4-10, 4-26
using CALL 4-11
using LINK 4-22
using the CSRL16J service 4-4
with a parameter list 4-9
with return 4-10
without control program assistance 4-7, 4-25
pass parameter
list 11-11
register 11-11
pass return address 4-8
PDS directory entry
AMODE indicator 4-1
RMODE indicator 4-1, 4-2
percolate
See also recovery
definition in a recovery environment 8-9
permanent object
accessing an existing object 16-10
creating a new object 16-10
data transfer 16-3
data-in-virtual object, relationship 16-1
defining a view 16-11
defining multiple views 16-14
definition 16-1
extending the size 16-14
functions supported for 16-6
identifying 16-9
mapping a scroll area to a permanent object, example 16-4
mapping with no scroll area, example 16-3
new object, creating 16-10
obtaining
access to a permanent object, overview 16-8
access to a permanent object, procedure for 16-9
scroll area 16-10
overview of supported function 16-6
refreshing changed data 16-16
refreshing, overview 16-9
requirements for new object 16-10
saving changes, overview 16-8
saving interim changes 16-15
size, maximum 16-1
specifying new or old status 16-10
permanent object *(continued)*
specifying type of access for an existing object 16-10
structure 16-1
terminating access to a permanent object 16-19
updating on DASD 16-17
PGLOAD macro
page-ahead function 18-3
use 18-2
PGOUT macro
use 18-2
PGRLSE macro
use 18-1
PGSER macro
input 18-5
page-ahead function 18-3
protecting a range of virtual storage pages 18-3
use 18-2
PICA (program interruption control area)
pointer 7-2
purpose 7-2
restore previous 7-2
PIE (program interruption element)
purpose 7-2
planned overlay load module structure 4-6
planning
system logger services
writing an ENF event 48 listen exit 26-47
pointer-defined entry point address 4-4
policy
LOGR 26-7
system logger 26-7
post bit 6-2
POST macro
use 6-2
prepare to pass control
with return 4-10
without return 4-8
preparing for
system logger services
writing an ENF event 48 listen exit 26-47
primary mode
description 14-1
primary mode program
call a program 2-8
defined 2-1
linkage procedure 2-6
pass parameters 2-9
receive control from a caller 2-6
return control to a caller 2-7
priority
address space 3-2
assign 3-3
change 3-3
control program's influence 3-2
dispatch 3-2

priority *(continued)*
higher, when to assign 3-3
limit 3-2
subtask 3-2
task 3-2
private library 4-16
processor storage management 18-1, 18-15
program availability
increasing through recovery 8-3
program design 4-7
program interruption
cause 7-1
determine cause 7-3
determining the type 7-5
program management 4-1, 4-35
program mask 7-2
program object
definition 4-16
PSL (page service list) 18-5
PSW (program status word)
addressing mode bit 4-2, 4-3
PUT macro 24-3

Q

qname of a resource
purpose 6-10
QRYLANG macro 21-14
query service
running under an MVS/ESA system 22-2
running under an MVS/XA system 22-3
using 22-1

R

range list entry 19-6
RANGLIST parameter on HSPSERV 15-30, 15-36
RB (request block)
considerations for ESTAE-type recovery routines 8-27
relationship to ESTAE-type recovery routines 8-7
read
from a log stream 26-34
log data in data set format
LOGR subsystem 26-49
LOGR subsystem
eligible applications 26-49
read from a standard hiperspace 15-30, 15-35
read operation
for standard hiperspace 15-29, 15-30, 15-31
recovery 8-1, 8-51
ABEND dump
requesting 8-15
ABEND macro
choosing to issue 8-5
invoking RTM 8-45
STEP parameter 8-29
recovery *(continued)*
activated
state of recovery routine 8-5
advanced topics 8-45
advantages of providing 8-3
AMODE
ESTAE-type recovery routine 8-38
retry from an ESTAE-type recovery routine 8-39
ARR
choosing 8-8
ASC mode
ESTAE-type recovery routine 8-38
retry from an ESTAE-type recovery routine 8-39
ATTACH and ATTACHX macros
ASYNCH parameter 8-29
ECB parameter 8-46
ESTAI parameter 8-8, 8-20
ETXR parameter 8-46
PURGE parameter 8-28
STAI parameter 8-8
TERM parameter 8-30
attached task 8-8
authorization
ESTAE-type recovery routine 8-37
retry from an ESTAE-type recovery routine 8-39
availability
increasing 8-3
communication
between processes 8-4
means available to recovery routines 8-19
parameter area 8-7, 8-19
registers 8-19
SDWA 8-11, 8-19
SETRP macro 8-11
concepts 8-2
condition of the linkage stack
ESTAE-type recovery routine 8-38
retry from an ESTAE-type recovery routine 8-40
correcting errors 8-15
DAE
providing information 8-15
deactivated
state of recovery routine 8-5
deciding whether to provide 8-3
defined
state of recovery routine 8-5
designing into your program 8-1
dispatchable unit mode
ESTAE-type recovery routine 8-37
retry from an ESTAE-type recovery routine 8-39
DU-AL
ESTAE-type recovery routine 8-38
retry from an ESTAE-type recovery routine 8-39
dump
ABEND dump 8-15
checking for previous 8-15
requesting 8-15

recovery *(continued)*
environment
ESTAE-type recovery routine 8-37
factors other than register contents 8-37
register contents 8-31
retry from an ESTAE-type recovery routine 8-39
STAE and STAI routines 8-49
summary for ESTAE-type recovery routine and its retry routine 8-40
understanding 8-30
errors 8-4
examples 8-4
ESTAE and ESTAEX macros
0 parameter 8-7
ASYNCH parameter 8-29
CT parameter 8-7, 8-46
defining a recovery routine 8-7
OV parameter 8-46
PARAM parameter 8-20
PURGE parameter 8-28
TERM parameter 8-30
ESTAE and ESTAEX routine
activated 8-7
deactivated 8-7
defined 8-7
definition 8-8
no longer defined 8-7
ESTAE-type recovery routine
additional considerations 8-29
linkage stack considerations 8-27
outstanding I/Os 8-28
providing 8-7
RB considerations 8-27
RB relationship 8-7
return codes 8-34
special considerations 8-26
ESTAI routine
activated 8-8
deactivated 8-8
defined 8-8
definition 8-8
no longer defined 8-8
rules for retry RB 8-27
example
coded 8-42
mainline routine with one recovery routine 8-10
mainline routine with several recovery routines 8-11
footprints 8-14, 8-20
from software errors 8-1
general concepts 8-2
IEALSQRY macro 8-41
in control
state of recovery routine 8-5
interrupt status
ESTAE-type recovery routine 8-38
retry from an ESTAE-type recovery routine 8-39

recovery *(continued)*
mainline routine
definition 8-6
minimizing errors 8-15
multiple recovery routines 8-46
MVS-provided 8-3
no longer defined
state of recovery routine 8-6
no longer in control
state of recovery routine 8-5
not providing 8-4
outstanding I/O
restoring quiesced restorable I/O operations 8-28
outstanding I/Os
controlling 8-28
parameter area
accessing 8-19, 8-21
checking the contents 8-14
contents 8-19
footprints 8-14, 8-20
passing 8-7, 8-19, 8-20
setting up 8-19
percolate
compared with retry 8-16
definition 8-9
program availability
increasing 8-3
program mask
ESTAE-type recovery routine 8-38
retry from an ESTAE-type recovery routine 8-40
quiesced restorable I/O operation
restoring 8-28
recovery routine
choosing 8-7
definition 8-6
nested 8-47
objectives 8-12
options 8-9
order of control 8-9
percolating 8-18
providing 8-1
providing recovery for a recovery routine 8-47
retrying 8-16
states 8-5
summary of states 8-9
writing 8-11
recursion
avoiding 8-14
definition 8-14
register contents 8-31
entry to a recovery routine 8-32
entry to a retry routine 8-35
restoring 8-17
return from a recovery routine 8-34
STAE or STAI retry routines 8-51
STAE routine 8-49

recovery *(continued)*
register contents *(continued)*
summary of where to find information 8-32
retry
compared with percolate 8-16
definition 8-9
retry point
definition 8-6
retry routine
definition 8-6
description 8-17
routines in a recovery environment
definition 8-6
interaction 8-9
mainline routine 8-6
recovery routine 8-6
retry routine 8-6
RTM 8-1
invoking 8-45
SDWA
accessing 8-21
accessing the SDWARC1 DSECT 8-23
checking important fields 8-13
directly manipulating fields 8-15
freeing 8-17
IHASDWA mapping macro 8-21
summary of important fields 8-23
updating 8-11, 8-21
updating through SETRP macro 8-14
updating through VRADATA macro 8-14
using 8-21
SDWA storage key
ESTAE-type recovery routine 8-37
retry from an ESTAE-type recovery routine 8-39
serviceability data
providing 8-4
saving 8-14
updating the SDWA 8-14
SETRP macro
communicating recovery options to the system 8-22
COMPCOD parameter 8-22
DUMP parameter 8-15
FRESDWA parameter 8-17, 8-35
indicating percolation 8-18
indicating retry 8-16
RC parameter 8-16, 8-18
REASON parameter 8-22
RECPARM parameter 8-14
REMREC parameter 8-18
RETADDR parameter 8-16
RETREGS parameter 8-17, 8-35
supplying a retry address 8-16
updating the SDWA 8-11, 8-22
STAE macro
0 parameter 8-7
CT parameter 8-7
recovery *(continued)*
STAE macro *(continued)*
defining a recovery routine 8-7
STAE retry routine 8-50
STAE routine
return codes 8-49
using 8-48
work area 8-49
STAI retry routine 8-50
STAI routine
return codes 8-49
using 8-48
work area 8-49
state of recovery routine
activated 8-5
deactivated 8-5
defined 8-5
in control 8-5
no longer defined 8-6
no longer in control 8-5
system logger 26-53
application failure 26-53
coupling facility failure 26-55
coupling facility full 26-57
DASD log data set space fills 26-58
log stream damage 26-58
peer connector 26-54
staging data set full 26-57
system failure 26-54
system logger address space failure 26-54
unrecoverable I/O errors 26-59
task 8-7
validity checking of user parameters 8-4
VRADATA macro
updating the SDWA variable recording area 8-14
writing recovery routines 8-11
checking for the SDWA 8-13
checking important fields in the SDWA 8-13
checking the parameter area 8-14
comparison of retry and percolate 8-16
correcting or minimizing errors 8-15
determining if the recovery routine can retry 8-16
determining the first recovery routine to get control 8-13
determining why the routine was entered 8-13
establishing addressability to the parameter area 8-13
locating the parameter area 8-13
providing information for DAE 8-15
requesting a dump 8-15
saving serviceability data 8-14
saving the return address to the system 8-12
recursion
avoiding in recovery 8-14
definition in recovery 8-14

reenterable
load module 4-21, 4-25, 11-11
macro 11-11
reenterable code
use 11-2, 11-5, 11-11
reenterable load module
use 11-3, 11-11
reference pattern service 18-5, 18-15
reference unit in reference pattern service
choosing 18-8
definition 18-8
reference unit in reference pattern services
definition 18-8
REFPAT macro
example 18-14
use 18-5
using 18-10
refresh changed data in an object 16-16
refreshable module 11-13
REGION system parameter 11-1
register
altering the content 7-5
provide a save area 2-2
save 2-2
register 1
pass parameters 4-8
register 14
use 4-10
when to restore 4-8
register 15
use 4-8
registers 2-12 4-10
release
data space and hiperspace storage 15-4
data space storage 15-16
rules 15-16
hiperspace storage 15-34
rules for 15-34
resource 6-14
virtual storage 18-1
RELEASE parameter on HSPSERV macro 15-31
release storage in data spaces
rules 15-16
remove
entry from access list 14-10
REPLACE option for a window 16-12
reply to WTOR message 20-9
request
dump 9-1
request for resource
limit concurrent 6-12
requirements for window service
DFP requirement 16-10
SMS requirement 16-10
RESERVE macro 6-17
compared to the ENQ macro 6-17
RESERVE macro *(continued)*
using for shared resources 6-17
using with the DEQ macro 6-17
resource
cleaning up 8-1
collecting information 6-17
control 6-1
duplicate request 6-14
global 6-11
local 6-11
name list 6-11
naming 6-10
process request 6-12
protecting
via serialization 6-9
release 6-14
releasing through recovery 8-3
requesting
conditionally 6-14
exclusive control 6-12
pair 6-16
shared control 6-12
unconditionally 6-14
scope 6-17
serially reusable
use 6-9
types that can be shared 6-11
resource manager
and delete requests 26-39
connecting as 26-25
exit
IXGCONN service 26-28
using with ENF event code 48 26-26
resource protection
via serialization 6-9
resource queue
extracting information 6-17
resource serialization
avoiding an interlock 6-15
requesting exclusive control 6-12
requesting shared control 6-12
responsibility count for a loaded module 11-13
restore
data space 15-25
hiperspace 15-41
PICA (program interrupt control area) 7-2
registers upon return 4-13
RETAIN option for a window 16-12
retry
See also recovery
definition in recovery environment 8-9
retry point
definition 8-6
retry routine
definition 8-6
ensure correct level of the linkage stack 8-41

return
control
in a dynamic structure 4-26
in a simple structure 4-13
return address
pass 4-8
return attributes function of unit verification service A-2
return code
analyzing 4-12
establish 4-14
for cell pool service 12-6
using 4-12
return group ID function of unit verification service A-2
return look-up value function of unit verification service A-2
RETURN macro
use 4-13, 4-14
return UCB addresses function of unit verification service A-2
return unit name function of unit verification service A-2
returned storage
specify subpool A-2
reusability attributes of a load module 4-25
reusable module 4-21
reuse of a save area 4-10
RIB (resource information block)
used with GQSCAN macro 6-18
RMODE program attribute
indicator in PDS entry 4-1
purpose 4-1
specifying
in source code 4-1
using linkage editor control card 4-1
use 4-16
value 4-2
route
code 20-6
message 20-6
route message 20-7
route the message 20-7
RTM (recovery termination manager)
See also recovery
MVS component that handles recovery 8-1
run length encoding 22-1
run-time message file
updating 21-13

S

SAC instruction
example 14-9
use 14-1
safe import point
IXGIMPRT service 26-41
IXGQUERY service 26-41
SAR instruction
description 14-8
example 14-8
save
data space 15-25
hiperspace 15-41
save area
example of using a 2-3, 2-5
how to tell if used 4-14
pass address 4-8
reuse 4-10
using a caller-provided save area 2-4
who must provide 2-2
save interim changes to a permanent object 16-15
SAVE macro
example of using 2-4
use 4-30
scope
ALL parameter value on GQSCAN macro 6-20
STEP parameter value on GQSCAN macro 6-17, 6-20
SYSTEM parameter value on GQSCAN macro 6-17, 6-20
SYSTEMS parameter value on GQSCAN macro 6-17, 6-20
scope of a resource
changing 6-11
STEP, when to use 6-11
SYSTEM, when to use 6-11
SYSTEMS, when to use 6-11
use 6-11
SCOPE parameter on DSPSERV 15-8
scroll area
data transfer 16-3
definition 16-2
mapping a scroll area to DASD, example 16-4
mapping a window to a scroll area, example 16-4
obtaining a scroll area 16-10
refreshing a scroll area 16-16
saving changes in, overview 16-8
saving interim changes in a 16-15
storage used for 16-2
updating a permanent object from a scroll area 16-17
updating DASD from, overview 16-8
use 16-2
SCROLL hiperspace 15-26
SDB (structured data base) format
description 10-3
SDWA (system diagnostic work area)
providing symptom data for a dump 9-6
SDWAARER field 8-24
SDWAARSV field 8-25

SDWA (system diagnostic work area) *(continued)*
SDWACID field 8-15, 8-26
SDWACLUP bit 8-16, 8-26
SDWACMPC field 8-13, 8-23
SDWACOMU field 8-26
SDWACRC field 8-13, 8-24
SDWAEAS bit 8-15, 8-26
SDWAEC1 field 8-24
SDWAEC2 field 8-24
SDWAGRSV field 8-24
SDWAINTF bit 8-24, 8-25
SDWALNTH field 8-26
SDWALSLV field 8-26, 8-41
SDWAMLVL field 8-15, 8-26
SDWAPARM field 8-13, 8-23
SDWAPERC bit 8-14, 8-26
SDWARPIV bit 8-23
SDWARRL field 8-15, 8-26
SDWASC field 8-15, 8-26
SDWASPID field 8-25
SDWASR00 field 8-18
SDWASRSV field 8-25
SDWATEAR field 8-26
SDWATEAV bit 8-26
SDWATEIV bit 8-26
SDWATRAN field 8-26
SDWAXFLG field 8-24, 8-25
search for a load module 4-17, 4-21
areas/libraries searched 4-18
limiting 4-18
order 4-18
serially reusable
module
obtaining a copy 4-21
pass control 4-25
resource
using 6-9, 6-17
serviceability data
providing through recovery 8-4
saving in the SDWA 8-14
set up
addressability to a data space
example 14-9
system logger configuration 26-47
SETRP macro
COMPCOD parameter 8-22
DUMP parameter 8-15
FRESDWA parameter 8-17, 8-35
RC parameter 8-16, 8-18
REASON parameter 8-22
RECPARM parameter 8-14
REMREC parameter 8-18
RETADDR parameter 8-16
RETREGS parameter 8-17, 8-35
updating the SDWA 8-11
share subpools 11-8, 11-10
shared pages 19-1
shared resource control
through the ENQ macro 6-12
through the RESERVE macro 6-17
shared standard hiperspace
definition 15-28
shared storage
with IARVSERV macro 19-1
sharing data in virtual storage
summary 1-3
sharing data in virtual storage (IARVSERV macro) 19-1
sharing data spaces 15-20
simple load module structure 4-6, 4-7
SMS requirement for window service 16-10
SNAP data control block 9-9
SNAP dump
index 9-9
requesting 9-9
SNAP macro
use 9-9
SNAPX macro
use 9-9
software error
recovering 8-1
specify subpool for returned storage A-2
SPIE (specify program interruption exit) environment
addressing mode 7-1
adjusting 7-2
canceling 7-2
definition 7-2
reestablishing 7-2
SPIE macro
addressing mode restriction 7-1
use 7-1, 7-2
STAE macro
0 parameter 8-7
CT parameter 8-7
defining a recovery routine 8-7
STAE routine
using 8-48
staging data sets 26-9
formatting 26-34
full condition 26-57
storage limit reached 26-34
system logger 26-9
STAI routine
using 8-48
STAM instruction
description 14-8
standard hiperspace
definition 15-26
example of creating 15-29
non-shared 15-28

standard hiperspace *(continued)*
read and write operation 15-30
shared 15-28
use 15-27
START parameter on DSPSERV 15-16, 15-34
STATUS macro 3-3
STCKCONV macro
using 20-2
STCKSYNC macro
using 20-1
step library
reason for limiting size 4-20
use 4-16
STIMER macro
using 20-2
STIMERM macro
using 20-2
STOKEN parameter on ALESERV 14-9
STOKEN parameter on DSPSERV 14-9, 15-8, 15-28
STOKEN parameter on HSPSERV 15-30
storage
freshly obtained 13-5
managing data space 15-17
subpool returned storage A-2
storage available for data space and hiperspace 15-3
STORAGE macro
OBTAIN request
example 15-36
use 11-1, 11-2, 11-4, 11-5
storage request
explicit 11-1
implicit 11-1
storage subpool 11-6
structure of a data object 16-1
subpool
characteristic 11-8
creating 11-9
handling 11-6
in task communication 11-10
ownership 11-9
sharing 11-8, 11-10
storage key for 11-8
transferring ownership 11-10
subpool release
definition 11-6
substitution token 21-4
subtask
communication with tasks 3-3
control 3-1
create 3-1
priority 3-2
starting 3-3
stopping 3-3
terminating 3-4, 6-2
summary dump 9-9
suppression
of dumps 9-2
symbol substitution
summary 1-3
symptom
provided by a recovery routine 9-6
required for dump suppression 8-22
symptom dump 9-2
symptom record 10-3
description 10-1
SYMRBLD macro
building a symptom record 10-1, 10-3
SYMREC macro
symptom recording 10-1
SYSABEND ABEND dump 9-1
SYSMDUMP ABEND dump 9-1
sysplex environment
communication 20-10
SYSSTATE macro
example 14-10, 14-11
use 4-21, 9-9, 14-10
system convention for parameter list 4-8
SYSTEM inclusion resource name list 6-11
system log
writing 20-11
system logger
ANSAREA parameter 26-18
answer area 26-18
size 26-18
answer area mapping
IXGANSAA macro 26-18
authorization requirements 26-11
complete exit 26-13
component 26-6
configuration 26-4
illustration 26-5, 26-6
connecting to and disconnection from a log stream 26-23
couple data set 26-7
DASD-only log stream 26-4
definition 26-1
delete data from a log stream
illustration 26-39
IXGDELET service 26-38
disconnect service 26-27
duplexing
DASD-only 26-9
duplexing log data 26-9
DASD-only 26-9
from a coupling facility log stream 26-9
entry-to-element ratio 26-8
IXGBRWSE service
REQUEST=READBLOCK 26-35
REQUEST=READCURSOR 26-35
REQUEST=RESET 26-35
REQUEST=START 26-35

system logger *(continued)*
IXGBRWSE service *(continued)*
searching for a log block by time stamp 26-36
with IXGDELET service 26-37
with IXGWRITE service 26-37
log data on the coupling facility 26-3
log data sets
allocation 26-4, 26-8
SMS management of 26-8
log stream 26-1, 26-2
illustration 26-1
LOGR couple data set 26-7
LOGR policy 26-7
LOGR subsystem 26-49
eligible applications 26-49
JCL for 26-51
using 26-51
managing the LOGR policy
IXGINVNT service 26-20
model log stream
example 26-21
peer connector
in recovery 26-54
planning
writing an ENF event 48 listen exit 26-47
policy 26-7
read
log data in data set format 26-49
read from a log stream
browse cursor 26-35
browse session 26-34
browse token 26-34
IXGBRWSE service 26-34
recovery 26-53
application failure 26-53
coupling facility failure 26-55
coupling facility full 26-57
DASD log data set space fills 26-58
log stream damage 26-58
staging data set full 26-57
system failure 26-54
system logger address space failure 26-54
unrecoverable I/O errors 26-59
resource manager exit 26-28
searching for a log block by time stamp
illustration 26-37
services
overview 26-10
staging data sets 26-9
formatting 26-34
storage limit reached 26-34
status changes
ENF event code 48 26-19
summary 1-3
user data for a log stream 26-27
writing to a log stream
IXGWRITE service 26-32
system logger application
ENF event code 48 26-19
example 26-1
system logger configuration
set up 26-47
system logger services 26-1
authorization requirements 26-11
gaps in the log stream 26-15
IXGBRWSE service 26-34
browse cursor 26-35
browse session 26-34
browse token 26-34
REQUEST=READBLOCK 26-35
REQUEST=READCURSOR 26-35
REQUEST=RESET 26-35
REQUEST=START 26-35
searching for a log block by time stamp 26-36
with IXGDELET service 26-37
with IXGWRITE service 26-37
IXGCONN service 26-22
allocating coupling facility space at connection 26-24
authorization requirements 26-24
connect process and staging data sets 26-24
disconnect from a log stream 26-27
ENF event code 48 and 26-27
import connection 26-26
resource manager connection 26-25
user data for a log stream 26-27
using ENF event 48 with 26-26
write connection 26-26
IXGDELET service 26-38
illustration 26-39
IXGWRITE service 26-32
BUFFALET parameter 26-32
BUFFKEY parameter 26-32
BUFFLEN parameter 26-32
committing data 26-33
coupling facility structure storage limit reached 26-33
log block buffer 26-32
sequence of log blocks 26-32
staging data set is formatting 26-34
staging data set storage limit reached 26-34
mode parameter 26-11
MODE=ASYNCNORESPONSE parameter 26-12
MODE=SYNC parameter 26-12
MODE=SYNCECB parameter 26-12
MODE=SYNCEXIT parameter 26-12
overview 26-10
preparing to use
writing an ENF event 48 listen exit 26-47
searching for a log block by time stamp
illustration 26-37
synchronous and asynchronous processing 26-11
system logger applications 26-1

system resource
releasing through recovery 8-3
system-generated PICA 7-4
SYSUDUMP ABEND dump 9-1

T

target program
definition 2-1
task
advantage of creating additional 3-1
communication with subtasks 3-3
create 3-1
library, establishing 4-17
priority, affect on processing 3-2
synchronization 6-2
TASKLIB parameter of ATTACH 4-17, 4-18
tasks in a job step
illustration 3-4
TCB (task control block)
address 3-1
remove 3-4
temporary object
accessing a temporary object 16-9
creating a temporary object 16-9
data transfer 16-3
defining a view 16-11
defining multiple views 16-14
definition 16-1
extending the size 16-14
functions supported for 16-7
initialized value 16-3
mapping a window, example 16-4
obtaining
access to a temporary object, overview 16-8
access to a temporary object, procedure for 16-10
scroll area 16-10
overview of supported function 16-7
refreshing changed data 16-16
refreshing, overview 16-9
saving changes, overview 16-8
size of, maximum 16-1
specifying the object size 16-10
storage used for 16-2
structure 16-1
terminating access to a temporary object 16-19
updating a temporary object 16-16
terminate access to an object 16-19
terminate view of an object 16-17
test return codes 4-12
time interval
example of using 20-3
TIME macro
using 20-1
time of day and date
obtaining 20-1
timer synchronization
checking 20-1
TIMEUSED macro
using 20-4
TOD (time-of-day) clock
checking for synchronization with ETR 20-1
converting value 20-2
obtaining content 20-1
token
used with GQSCAN macro 6-18
TRANMSG macro 21-14
Transaction dump 9-1
transfer data between hiperspace and address space 15-29
translate message 21-1
TTIMER macro
using 20-2

U

UCB (unit control block)
obtaining device information 23-3
scanning 23-2
UCBINFO macro 23-3
UCBSCAN macro 23-2
unit name A-2
device class A-3
is a look-up value function of unit verification service A-2
unit verification service
description A-1
examples A-13
functions
check groups A-1, A-5
check units A-1, A-5
convert device type to look-up value A-2, A-10
indicate unit name is a look-up value A-2, A-8
return attributes A-2, A-11
return group ID A-2, A-8
return look-up value A-2, A-9
return unit control block (UCB) address A-2, A-7
return unit name A-2, A-6
return unit names for a device class A-3, A-12
specify subpool for returned storage A-2, A-11
IEFEB4UV routine A-3
input and output A-3
data structure A-4
FLAGS parameter field A-3
parameter list required A-3
purpose A-1
requesting multiple functions A-13
update a permanent object on DASD 16-17
update a temporary object 16-16

UPDTMPB macro 21-17
use an entry to an access list
example 14-9
use count 4-23
use count for a loaded module 4-27
use data spaces efficiently 15-23
use of data space and hiperspace 15-3
user list 13-6, 13-15, 13-18
use
access 13-8
identify 13-7
map 13-10
reset 13-18
savelist 13-17

V

V-type address constant
use to pass control 4-10
V=R (virtual=central) storage
allocation 18-1
VERBEXIT DAEDATA subcommand
indicating why dump was not suppressed 9-8
verification
device A-1
version record
format 21-4
VFPM parameter list
format and content B-4
notes B-4
virtual fetch service
build function B-1
BUILD request B-1, B-5
considerations when using B-2
defined B-1
description B-1
dumping the managed modules B-4
find function B-2
FIND request B-1
functions B-1
get function B-2
GET request B-1
parameter list B-2
programming considerations B-4
for a BUILD request B-5
for a FIND request B-5
for a GET request B-5
parameters B-4
programming conventions B-4
requesting dumps B-4
starting virtual fetch B-1
using a cataloged procedure B-1
VFPM parameter list B-2
VIO data set B-1
virtual storage
controlling 11-6
virtual storage *(continued)*
explicit requests for 11-1
freeing 11-13
implicit requests for 11-11
loading 18-2, 18-3
obtaining via CPOOL 11-6
page-ahead function 18-3
paging out 18-3
releasing 18-1, 18-2
sharing with IARVSERV macro 19-1
specifying the amount allocated to a task 11-1
subpool 11-6
using efficiently 11-1
virtual storage window 13-1, 13-5
VRADATA macro
to customize dump suppression 9-6
using in a recovery environment 8-14
VSL (virtual subarea list) 18-4
VSM (virtual storage management) 11-1, 11-14

W

wait
bit 6-2
condition 6-2
long 6-3
WAIT macro
use 6-2
ways that window services can map an object 16-3
window
affect of terminating access to an object 16-19
blocks to be viewed, identifying 16-13
changing a view in a window 16-17
changing the view, overview 16-9
data to be viewed, identifying 16-13
defining>
multiple windows 16-14
window disposition 16-12
window reference pattern 16-12
window, overview 16-8
windows with overlapping view 16-15
definition 16-1
identifying a window 16-11
identifying blocks to be viewed 16-13
mappping
multiple objects, example 16-5
to a window, example 16-3
to multiple windows, example 16-5
refreshing a window 16-16
REPLACE option 16-12
RETAIN option 16-12
size 16-11
storage for 16-11
terminating a view in a window 16-17
updating a permanent object from a window 16-17
use 16-1

window service
introduction 16-1
overview 16-1
use 16-8
window services
functions provided 16-2
overview 16-1
services provided 16-2
using window services 16-8
ways to map an object 16-3
work area
used by data compression service 22-1
used by data expansion service 22-1
write
to the operator with reply 20-5
to the operator without reply 20-8
to the programmer 20-10
to the system log 20-11
write connection
IXGCONN service
IMPORTCONNECT parameter 26-26
WRITE macro 24-3
write message 20-5
write operation
for standard hiperspace 15-29
write programs in AR mode 14-5
write to a standard hiperspace 15-30, 15-35
writing
to a log stream 26-32
WTL macro
writing to the system log 20-11
WTO macro
descriptor code for 20-7
example 20-8
MLWTO (multiple-line) form 20-6
single-line form 20-6
use 20-5
WTOR macro
example 20-9
use 20-5

X

X-macro
definition 14-11
rules for using 14-11
XCTL macro
addressing mode consideration 4-16
lowering the responsibility count 11-13
use 4-16, 4-26, 4-27
use with branch instructions, danger 4-27
XCTLX macro
use 4-26, 4-27

Communicating Your Comments to IBM

OS/390
MVS Programming: Assembler Services
Guide

Publication No. GC28-1762-06

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. 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.

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.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:

  1-(914)-432-9405
- If you prefer to send comments electronically, use this network ID:

  mhvrcfs@us.ibm.com

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies.

Readers' Comments — We'd Like to Hear from You

OS/390
MVS Programming: Assembler Services Guide

Publication No. GC28-1762-06

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

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.

Name _____ Address _____

Company or Organization _____

Phone No. _____

Cut or Fold
Along Line

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

IBM Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY 12601-5400

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Program Number: 5647-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC28-1762-06