z/OS
2.4

MVS Programming: Extended Addressability Guide

SA23-1394-40

Note

Before using this information and the product it supports, read the information in “Notices” on page 237.

This edition applies to Version 2 Release 4 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2021-06-21

Contents

Figures ix

Tables xiii

About this document xv
Who should use this document xv
How this document is organized xv
How to use this document xvi
z/OS information xvi

How to send your comments to IBM xvii
If you have a technical problem xvii

Summary of changes xix
Summary of changes for z/OS Version 2 Release 4 xix
Summary of changes for z/OS Version 2 Release 3 xix
Summary of changes for z/OS Version 2 Release 2 xx

Chapter 1. An introduction to extended addressability 1
Basic concepts 3
Asynchronous cross memory communication 3
Synchronous cross memory communication 3
Access register ASC mode 3
Data-in-Virtual 4
Virtual lookaside facility 4
Data spaces and hiperspaces 4
Basic decision: data space or hiperspace 5
What can a program do with a data space or a hiperspace? 5
What are the differences? 6
Which one should your program use? 8
Choosing VIO instead of a data space or a hiperspace 9

Chapter 2. Linkage stack 11
Linkage stack considerations for asynchronous exit routines 12
Instructions that add and remove a linkage stack entry 12
Branch and stack (BAKR) instruction 12
Program return (PR) instruction 12
Example of using the linkage stack 13
Contents of the linkage stack entry 14
Instructions that manipulate the contents of a linkage stack entry 14
Extract stacked registers (EREG) instruction 14
Extract stacked state (ESTA) instruction 15
Modify stacked state (MSTA) instruction 16
Expanding a linkage stack to a specified size 16
Relationship between the linkage stack and ESTAE-type recovery routines 16
Dumping the contents of the linkage stack 17

Chapter 3. Synchronous cross memory communication 19
When should you use synchronous cross memory communication? 19

Terminology, macros, and assembler instructions....19
Cross memory terminology....19
Macros used for synchronous cross memory communication....21
Instructions used for cross memory communication....22
An overview of cross memory communication....22
PC routines....22
Summary of cross memory communication....25
The cross memory environment....26
Entry tables....26
Linkage tables....26
The PC number....26
Program authorization — the PSW-key mask (PKM)....27
Address space authorization....27
Considerations before using cross memory....30
Environmental considerations....31
Restrictions....31
Requirements....31
Establishing cross memory communication....31
Making a PC routine available to all address spaces....32
Making a PC routine available to selected address spaces....33
Examples of how to establish a cross memory environment....35
Example 1 - Making services available to selected address spaces....36
Example 2 - Making services available to all address spaces....43
Example 3 - Providing non-space switch services....45
PC linkages and PC routine characteristics....45
PC linkage capabilities....45
Defining a PC routine....45
PC routine requirements....48
Linkage conventions....49
Resource Management....52
Reusing ASIDs....53
Reassigning LXs when the LX reuse facility is enabled....56
Reusing AXs and EAXs....57
PC Routine Loading Recommendations....57
Accounting Considerations....57
Recovery Considerations....57

Chapter 4. Using the 64-bit address space....59
What is the 64-bit address space?....59
Why would you use virtual storage above the bar?....61
Memory management above the bar....61
Memory objects....61
Limiting the use of private memory objects....62
Using large pages....67
Using assembler instructions in the 64-bit address space....67
64-bit binary operations....68
64-bit addressing mode (AMODE)....69
Using a memory object....71
IARV64 macro services....73
Protecting storage above the bar....75
Tagging 64-bit memory objects for data privacy....75
Creating private memory objects....76
Freeing a private memory object....77
An example of creating, using, and freeing a private memory object....78
Creating shared memory objects....79
Freeing a shared memory object....83
Creating common memory objects....84

Freeing a common memory object ... 86
Fixing the pages of a memory object ... 87
Discarding data in a memory object ... 88
Releasing the physical resources that back pages of memory objects ... 88
Creating guard areas and changing their sizes ... 88
Listing information about the use of virtual storage above the bar ... 91
Changing the attributes of storage within a memory object ... 91
Dumping 64-bit common memory objects ... 91

Chapter 5. Using access registers ... 93
Using access registers for data reference ... 93
A comparison of data reference in primary and AR mode ... 95
Coding instructions in AR mode ... 96
Manipulating the contents of ARs ... 97
Access lists ... 98
Types of access lists ... 98
Types of access list entries ... 102
Special ALET values ... 103
Special ALET Values at a Space Switch ... 103
Loading the Value of Zero into an AR ... 104
The ALESERV macro ... 105
Setting up addressability to an address/data space ... 105
Adding an entry to an access list ... 106
Example of Adding an Access List Entry for a Data Space ... 107
Example of adding an access list entry for an address space ... 107
Obtaining and passing ALETs and STOKENs ... 108
Examples of establishing addressability to data spaces ... 108
Adding an Entry for the Primary Address Space to the DU-AL ... 112
Using the ALET for the Home Address Space ... 113
Deleting an entry from an access list ... 113
Example of deleting a data space entry from an access list ... 114
Example of deleting an address space entry from an access list ... 114
ALET reuse by the system ... 114
EAX-authority to an address space ... 115
Setting the EAX value ... 117
Procedures for establishing addressability to an address space ... 118
Changing an EAX value ... 120
Freeing an EAX value ... 120
Checking the authority of callers ... 120
Obtaining storage outside the primary address space ... 122
What access lists can an asynchronous exit routine use? ... 123
Issuing MVS macros in AR mode ... 123
Passing parameters to MVS macros in AR mode ... 125
Formatting and displaying AR information ... 125

Chapter 6. Creating and using data spaces ... 127
Referencing data in a data space ... 127
Relationship between the data space and its owner ... 128
Scope=single, scope=all, and scope=common data spaces ... 128
Rules for creating, deleting, and using data spaces ... 129
Example of the rules for accessing data spaces ... 130
Summary of rules for creating, deleting, and using data spaces ... 131
Creating a data space ... 133
Choosing the name of the data space ... 134
Specifying the size of the data space ... 134
Identifying the origin of the data space ... 136
Example of creating a data space ... 136

Protecting data space storage ... 137
Creating a data space of DREF storage ... 137
Establishing addressability to a data space ... 138
Example of establishing addressability to a data space ... 138
Managing data space storage ... 138
Managing data space storage across a checkpoint/restart operation ... 138
Limiting data space use ... 139
Serializing use of data space storage ... 139
Examples of moving data into and out of a data space ... 139
Using callable cell pool services to manage data space areas ... 141
Extending the current size of a data space ... 143
Deleting a data space ... 144
Example of creating, using, and deleting a data space ... 144
Creating and using SCOPE=COMMON data spaces ... 145
Attaching a subtask and sharing data spaces with it ... 147
Sharing data spaces among problem state programs with PSW key 8 through F ... 148
Mapping a data-in-virtual object to a data space ... 148
Paging data space storage areas into and out of central storage ... 150
Releasing data space storage ... 151
How SRBs use data spaces ... 151
Obtaining the TCB identifier for a task (ttoken) ... 154
Example of an srb routine using a data space ... 154
Dumping storage in a data space ... 156
Using data spaces efficiently ... 157

Chapter 7. Creating and using hiperspaces ... 159
Managing hiperspace storage ... 160
Limiting hiperspace use ... 160
Managing hiperspace storage across a checkpoint/restart operation ... 160
Relationship between the hiperspace and its owner ... 161
Serializing use of hiperspace storage ... 161
Standard and expanded storage only hiperspaces ... 161
Standard hiperspace ... 161
Expanded storage only hiperspaces ... 162
Summary of the differences ... 163
Rules for creating, deleting, and using hiperspace ... 163
Creating a hiperspace ... 164
Choosing the name of the hiperspace ... 165
Specifying the size of the hiperspace ... 166
Protecting hiperspace storage ... 167
Identifying the origin of the hiperspace ... 168
Creating a non-shared or shared standard Hiperspace ... 168
Creating an expanded storage only Hiperspace ... 168
Accessing hiperspaces ... 169
How an ALET connects a program to a hiperspace ... 170
How problem state programs with PSW key 8 through F use a hiperspace ... 170
How supervisor state or PSW key 0 through 7 programs use hiperspaces ... 172
Obtaining an ALET for a hiperspace ... 174
Transferring data to and from a hiperspace ... 176
Read and write operations for standard hiperspaces ... 177
Read and write operations for expanded storage only hiperspaces ... 179
Obtaining improved data transfer to and from a hiperspace ... 180
Extending the current size of a hiperspace ... 190
Deleting a hiperspace ... 191
Releasing hiperspace storage ... 191
Using data-in-virtual with standard hiperspaces ... 192
Mapping a data-in-virtual object to a hiperspace ... 193

Using a hiperspace as a data-in-virtual object....195
How SRBs use hiperspaces....196

Chapter 8. Creating address spaces....197
Using the ASCRE macro to create an address space....197
Planning the characteristics of the address space....198
Identifying a procedure in SYS1.PROCLIB....199
The address space initialization routine....200
Writing an Initialization Routine....200
Establishing cross memory linkages....202
Passing a parameter list to the new address space....205
Providing an address space termination routine....205
Establishing attributes for address spaces....206
Deleting an address space....207
Example of creating and deleting an address space....207

Chapter 9. Creating and using subspaces....211
What is a subspace?....211
Deciding whether your program should run in a subspace....214
Benefits of subspaces....214
Limitations of subspaces....214
System storage requirements....215
Steps to manage subspaces....215
Updating the application server to use subspaces....217
Managing subspaces when performance is a priority....217
Managing subspaces when storage is a priority....217
Creating a single subspace....217
Determining whether subspaces are available on your system....218
Obtaining storage for subspaces....219
Making a range of storage eligible to be assigned to a subspace....220
Creating the subspaces....222
Establishing addressability to a subspace....223
Assigning storage to the subspaces....223
Branching to a subspace....224
Running a program in a subspace....225
Disassociating storage from the subspaces....227
Removing the subspace entry from the DU-AL....227
Deleting the subspace....228
Making storage ineligible to be assigned to a subspace....228
Releasing storage....228
Example of managing subspaces....228
Planning for recovery in a subspace environment....230
Planning for SPIE and ESPIE routines....231
Planning for ESTAE-type recovery routines and FRRs....231
Diagnosing errors in a subspace environment....232
Diagnosing 0C4 abends....232
Using IPCS to diagnose program errors in a subspace....232
RSM component trace....232
Requesting a dump....232

Appendix A. Accessibility....233
Accessibility features....233
Consult assistive technologies....233
Keyboard navigation of the user interface....233
Dotted decimal syntax diagrams....233

Notices....237

Terms and conditions for product documentation..... 238
IBM Online Privacy Statement..... 239
Policy for unsupported hardware..... 239
Minimum supported hardware..... 239
Programming Interface Information..... 240
Trademarks..... 240
Glossary..... 241

Index..... 247

Figures

1. Accessing data in a data space..... 7

2. Accessing data in a hiperspace..... 7

3. Example of using the linkage stack..... 13

4. Format of the information fields..... 15

5. Example of an ESTA instruction..... 15

6. Example of an MSTA instruction..... 16

7. PC routine invocation..... 24

8. Accessing data through the MVCP and MVCS instructions..... 25

9. PC instruction execution environment..... 29

10. PT and SSAR instruction execution environment..... 30

11. Using ETDEF to statically define entry table descriptors..... 38

12. Using ETDEF to dynamically define entry table descriptors..... 39

13. Linkage table and entry table connection..... 41

14. Calling sequence for a stacking PC routine..... 41

15. Calling sequence for a basic PC routine..... 42

16. Linkage and entry tables for a global service..... 44

17. Cross memory connections between address spaces..... 54

18. z/OS address space..... 60

19. Order of precedence in determining the effective MEMLIMIT value..... 64

20. z/OS R5 Address Space with the default shared area between 2-terabytes and 512-terabytes 80

21. A memory object eight megabytes in size..... 89

22. A memory object with an additional guard area..... 90

23. Example of an AR/GPR..... 94

24. Using an ALET to identify an address/data space..... 95

25. The MVC instruction in primary mode..... 96

26. The MVC Instruction in AR Mode..... 96

27. Comparison of addressability through a PASN-AL and a DU-AL..... 101

28. PASN-ALs and DU-ALs at a space switch..... 102

29. Special ALET values..... 103

30. Example 1: Adding an entry to a DU-AL..... 109

31. Example 1: Sharing a data space through DU-ALs..... 109

32. Example 2: Adding an entry to a PASN-AL..... 110

33. Example 2: Sharing a data space through the PASN-AL..... 111

34. Example 3: Sharing data spaces between two address spaces..... 112

35. Obtaining the ALET for the Primary Address Space..... 112

36. Using the ALET for the Home Address Space..... 113

37. Difference Between Public and Private Entries..... 116

38. Comparison of an AX and an EAX..... 118

39. Checking the Validity of an ALET..... 121

40. Example of Rules for Accessing Data Spaces..... 131

41. Example of Specifying the Size of a Data Space..... 136

42. Protecting storage in a data space..... 137

43. Example of Using Callable Cell Pool Services for Data Spaces..... 142

44. Example of Extending the Current Size of a Data Space..... 143

45. Example of Using a SCOPE=COMMON Data Space..... 146

46. Two programs sharing a SCOPE=SINGLE data space..... 147

47. Example of Mapping a Data-in-Virtual Object to a Data Space..... 149

48. Scheduling an SRB with an empty DU-AL and in a non-cross memory environment..... 152

49. Scheduling an SRB with a copy of the scheduling program's DU-AL and in the same cross memory environment..... 153

50. Example of scrolling through a standard hiperspace..... 162

51. Example of specifying the size of a hiperspace..... 167

52. Protecting storage in a hiperspace..... 168

53. A problem state program using a non-shared standard hiperspace..... 170

54. Example 1: An unauthorized program using a standard hiperspace..... 171

55. Example 2: An unauthorized program using a standard hiperspace..... 172

56. A supervisor state program using a non-shared standard Hiperspace..... 173

57. A supervisor state program using a shared standard hiperspaces..... 174

58. Illustration of the HSPSERV write and read operations..... 177

59. Example of creating a standard hiperspace and transferring data..... 178

60. Gaining fast data transfer to and from expanded storage..... 181

61. Example of extending the current size of a hiperspace..... 191

62. Example of mapping a data-in-virtual object to a hiperspace..... 194

63. A Standard hiperspace as a data-in-virtual object..... 195

64. Synchronization of the address space creation process..... 201

65. An example of a cross memory environment..... 202

66. An example of cross memory environment set by the ASCRE macro..... 203

67. The cross memory linkages set by the ASCRE macro..... 205

68. Illustration of address space that owns one subspace..... 212

69. Illustration of address space that owns two subspaces..... 213

70. Illustration of the range list..... 221

71. Illustration of GPR Contents in Event of Range List Error..... 222

Tables

1. Data requirements for VIO, data spaces, and hiperspaces....9

2. Difficulty of modifying an existing application....10

3. Example of an EREG instruction.... 14

4. Macros to issue for a PC routine to be available to all address spaces.... 32

5. Macros that must be issued for basic and stacking PC routines.... 33

6. Declared storage for cross memory examples....35

7. Comparing tasks and concepts for memory objects: Below the bar and above the bar....72

8. IARV64 service requests and rules for programs working with memory objects....73

9. IARV64 services valid for private, shared, and common memory objects....75

10. Base and index register addressing in AR mode....97

11. Functions of the ALESERV Macro.... 105

12. Relationship between the CHKEAX and ACCESS parameters on ALESERV....115

13. Creating, Deleting, and Using Data Spaces....132

14. Requirements for authorized programs using the DIV services with data spaces.... 149

15. Addressability for each type of invocation of the SCHEDULE macro.... 153

16. Comparison of standard and ESO hiperspaces....163

17. Creating, deleting, and using hiperspace....164

18. Hiperspaces that problem state programs with PSW 8 - F can access....172

19. Hiperspaces that supervisor state or PSW key 0 - 7 programs can use.... 173

20. Rules for adding access list entries for hiperspaces....175

21. Uses of hiperspaces and data-in-virtual.... 192

22. Requirements for authorized programs using the DIV services with hiperspaces....193

23. Planning considerations for the new address space.... 198

24. ATTR options for address spaces..... 206

25. System storage requirements when managing subspaces..... 215

26. Steps for creating, using, and deleting subspaces..... 215

27. How a server program manages single subspaces.....217

28. Storage attributes required for subspaces.....219

About this document

This document is intended for the programmer who writes programs with needs that extend beyond the boundaries of the address space in which the programs are dispatched. Specifically, the programs need to do one or more of the following:

- Execute in a multi-address space environment, interacting with programs running in other address spaces
- Use data in address spaces other than the primary
- Use data in data spaces and hiperspaces
- Create another address space.

Who should use this document

This document is intended for programmers who write programs that interact with MVS™ or with subsystems. The programs must be in supervisor state, or PSW key 0 - 7, or reside in APF-authorized libraries, except where otherwise noted. The document assumes that the reader understands system concepts and writes programs in assembler language.

System macros require High Level Assembler. For more information about assembler language programming, see High Level Assembler and Toolkit Feature in IBM Documentation (www.ibm.com/docs/en/hla-and-tf/1.6).

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

How this document is organized

This document is organized as follows:

- Chapter 1, “An introduction to extended addressability,” on page 1 describes the concepts behind a multiple-address environment in which the functions described in the document would be appropriate. It describes the reasons why a programmer might want to extend the addressability of a program beyond the boundaries of a program's primary address space. It also compares two kinds of data-only spaces: data spaces and hiperspaces.
- Chapter 2, “Linkage stack,” on page 11 describes an area that the system provides a program to save status information at a branch or a program call instruction. This section describes the linkage stack and the assembler instructions that cause the system to add and remove an entry and use the entry.
- Chapter 3, “Synchronous cross memory communication,” on page 19 describes cross memory functions.
- Chapter 4, “Using the 64-bit address space,” on page 59 describes how a program can use the address space virtual storage above the 2-gigabyte address. The chapter describes the rules for creating, freeing, and using those virtual storage areas.
- Chapter 5, “Using access registers,” on page 93 describes how a program can use the registers known as "access registers" to access data in address spaces and data spaces.
- Chapter 6, “Creating and using data spaces,” on page 127 describes how a program can ask the system for an area of virtual storage known as a "data space". The chapter describes the rules for creating, deleting, and using data spaces.
- Chapter 7, “Creating and using hiperspaces,” on page 159 describes how a program can ask the system for an area of virtual storage known as a "hiperspace". The chapter describes the rules for creating, deleting, and using hiperspaces.
- Chapter 8, “Creating address spaces,” on page 197 describes how a program can use the ASCRE macro to create an address space.

- Chapter 9, “Creating and using subspaces,” on page 211 describes how a program can use subspaces to prevent multiple application programs running in a single address space from overwriting each other.

How to use this document

This document is one of the set of programming documents 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 documents, see *z/OS Information Roadmap*.

z/OS information

This information explains how z/OS references information in other documents and on the web.

When possible, this information uses cross document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see *z/OS Information Roadmap*.

To find the complete z/OS® library, go to IBM Documentation (www.ibm.com/docs/en/zos).

How to send your comments to IBM

We invite you to submit comments about the z/OS product documentation. Your valuable feedback helps to ensure accurate and high-quality information.

Important: If your comment regards a technical question or problem, see instead "If you have a technical problem" on page xvii.

Submit your feedback by using the appropriate method for your type of comment or question:

Feedback on z/OS function
If your comment or question is about z/OS itself, submit a request through the IBM RFE Community (www.ibm.com/developerworks/rfe/).

Feedback on IBM® Documentation function
If your comment or question is about the IBM Documentation functionality, for example search capabilities or how to arrange the browser view, send a detailed email to IBM Documentation Support at ibmdocs@us.ibm.com.

Feedback on the z/OS product documentation and content
If your comment is about the information that is provided in the z/OS product documentation library, send a detailed email to mhvrcfs@us.ibm.com. We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

To help us better process your submission, include the following information:

- Your name, company/university/institution name, and email address
- The following deliverable title and order number: z/OS MVS Extended Addressability Guide, SA23-1394-50
- The section title of the specific information to which your comment relates
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive authority to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

If you have a technical problem or question, do not use the feedback methods that are provided for sending documentation comments. Instead, take one or more of the following actions:

- Go to the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Summary of changes

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

Summary of changes for z/OS Version 2 Release 4

The following information is new, changed, or deleted in z/OS Version 2 Release 4 (V2R4).

New

The following new information is added in this publication:

September 2020 refresh

- Information about CHANGEATTRIBUTE is added in Table 8 on page 73 and Table 9 on page 75 with APAR OA58289.
- The topic, "Changing the attributes of storage within a memory object" on page 91, is added with APAR OA58289.

Prior to September 2020 refresh

- The SENSITIVE parameter is added in Tagging 64-bit memory objects for data privacy, "GETSTOR request" on page 76, "GETSHARED request" on page 79, and "GETCOMMON request" on page 85 (APAR OA57633).
- The INORIGIN parameter is added in "GETSTOR request" on page 76.

Changed

The following information is changed in this publication:

June 2021 refresh

- Information about work units is updated in "Access lists" on page 98.

May 2021 refresh

- Programming considerations are updated in "Creating and using SCOPE=COMMON data spaces" on page 145.

September 2020 refresh

- "Limiting the use of private memory objects" on page 62 is updated.
- "Using large pages" on page 67 is updated.

Prior to September 2020 refresh

- "Example 1 - Making services available to selected address spaces" on page 36 is updated in "Setting Up" on page 36.

Summary of changes for z/OS Version 2 Release 3

The following information is new, changed, or deleted in z/OS Version 2 Release 3 (V2R3).

New

The following information has been added:

- Added a new section on "Authorized programs and subspaces" on page 226 for APAR OA54807.

- Added a new paragraph for MEMLIMIT under the “Limiting the use of private memory objects” on page 62 section.
- Added parameters SADMP, EXECUTABLE, LOCALSYSAREA, MEMLIIT, DETACHFIXED, and DUMP under the “GETSTOR request” on page 76 section.
- Added parameters SADMP and EXECUTABLE under the “GETCOMMON request” on page 85 section.

Changed

The following information has been changed:

- Information about the INORIGIN keyword for the IARV64 macro is added in “GETSTOR request” on page 76 for APAR OA56664.
- Information is updated in “Using the ASCRE macro to create an address space” on page 197.
- Information about subspaces has been updated in “Limitations of subspaces” on page 214 for APAR OA54807.
- Information about system-generated names has been updated in “Choosing the name of the data space” on page 134.
- Information about the save area has been updated in “Providing an address space termination routine” on page 205.

Summary of changes for z/OS Version 2 Release 2

The following information is new, changed, or deleted in z/OS Version 2 Release 2 (V2R2).

New

The following information has been added:

- Information about 64-bit common storage has been added in Chapter 4, “Using the 64-bit address space,” on page 59.

Changed

The following information has been changed:

- “Program authorization — the PSW-key mask (PKM)” on page 27 has been updated for improved clarity.
- “Creating guard areas and changing their sizes” on page 88 has been updated.

Chapter 1. An introduction to extended addressability

Over the years, MVS has changed in many ways. Two key areas of growth and change are addressability and integrity. The concept of an address space is an integral part of both. An address space, literally defined as the range of addresses available to a computer program, is like a programmer's map of the virtual storage available for code and data. An address space provides each programmer with access to all of the addresses available through the computer architecture.

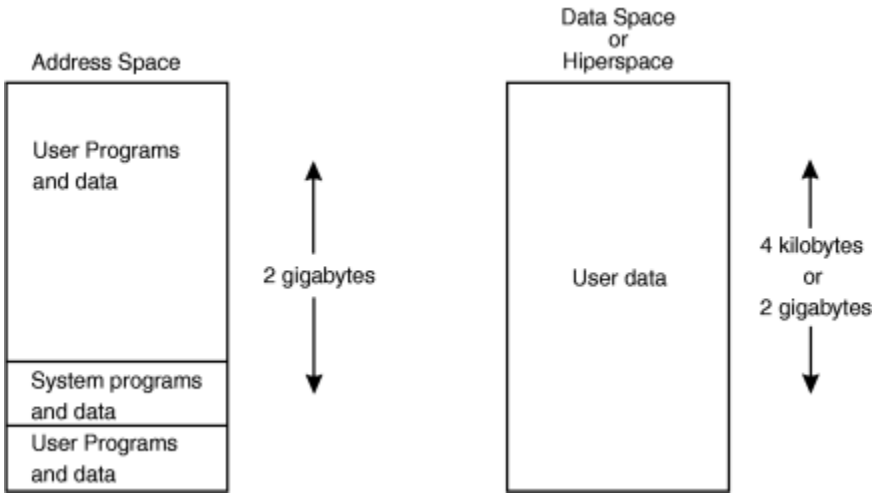
Because it maps all of the available addresses, an address space includes system code and data as well as user code and data. Thus, not all of the mapped addresses are available for user code and data. This limit on user applications was a major reason for System/370 Extended Architecture (370-XA) and MVS/XA. Because the effective length of an address field expanded from 24 bits to 31 bits, the size of an address space expanded from 16 megabytes to 2 gigabytes. An MVS/XA address space is 128 times as big as an MVS/370 address space.

A 2-gigabyte address space, however, does not, in and of itself, meet all of programmers' needs in an environment where processor speed continues to increase, where applications must support hundreds of users with instant response time requirements, and where businesses depend on quick access to huge amounts of information stored on DASD.

With z/OS, the MVS address space expands to a size so vast that we need new terms to describe it. Each address space, called a 64-bit address space, is 16 exabytes in size; an exabyte is slightly more than one billion gigabytes. The new address space has logically 2^{64} addresses. It is 8 billion times the size of the former 2-gigabyte address space that logically has 2^{31} addresses. The number is 16 with 18 zeros after it: 16,000,000,000,000,000,000 bytes, or 16 exabytes. If you are coding a new program that needs to store large amounts of data, a 64-bit address space might work for you. See Chapter 4, “Using the 64-bit address space,” on page 59.

If, however you need more than a large address space, other extended addressability techniques meet that need. Extended addressability allows programmers to extend the power of applications through the use of additional address spaces or data-only spaces. The data-only spaces that are available for your programs are called data spaces and hiperspaces. These spaces are similar in that both are areas of virtual storage that your program can ask the system to create. Their size can be up to 2 gigabytes, as your program requests. Unlike an address space, a data space or hiperspace contains only user data; it does not contain system control blocks or common areas. 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.



Both the architecture and the system protect the integrity of code and data within an address space. Various techniques, like storage protect key and supervisor state requirements, provide protection that is almost like a wall around an address space, and this wall is basically a good thing from the point of view of the work going on inside that individual address space.

The programming techniques that provide extended addressability permit programs to break through but still preserve the wall that protects the address space.

Whether your application is one that can use extended addressability depends on many factors. One basic factor is the amount of central, expanded, and auxiliary storage available at your installation to back up virtual storage. Extended addressability frequently requires additional amounts of virtual storage, which means that your installation must have sufficient central and auxiliary storage, and some of the techniques work most efficiently only when expanded storage is available.

The goals for the design of a particular application are equally important in the decision-making process. These goals might include:

- Performance. For an application with large numbers of online end users, achieving the best possible response time is always a significant design goal.
- Efficient use of system resources, such as storage, and efficient use of the DASD resources.
- Ability to randomly access very large amounts of data.
- Data integrity and isolation. Data in an address space is generally available to all tasks (or TCBs) running in that address space; access to data in a data space or hiperspace can be restricted. Code running in an address space can inadvertently overlay data; because of its isolation, data in a data space or hiperspace is less likely to be overlaid.
- Independence from individual device characteristics, from record-oriented processing, and from data management concerns in general. Extended addressability can allow an application to focus on controlling data as information in contrast to controlling data as records in data sets stored on DASD volumes.
- Reduction in the size and complexity of the programming effort required to develop a new application.

Achieving these goals depends to a very great extent on choosing a way to extend addressability that meets your needs. You need to understand, at a very high level, basic concepts related to each technique and how you might apply extended addressability to specific programming situations.

At the detailed technical level, extended addressability can mean learning new programming techniques, or new ways of applying existing techniques. At a higher level, extended addressability can open completely different solutions to programming problems. With extended addressability, virtual storage, backed by expanded storage, can become, conceptually, a high-performance medium for application data. It is also important to note that you should think of extended addressability techniques as ones you can use to modify existing applications as well as code new ones.

To use an example of how extended addressability can open up new solutions, assume you need to write an application to sort 5000 records.

If you can hold only 50 records in storage, you must use DASD for intermediate workfile processing.

If you can hold 500 records in storage, the solution is still the same, though it requires fewer I/O operations.

If you can hold all 5000 records in storage, the original solution still works, but it is now possible to devise a completely different solution, one, for example, that does not depend on a DASD workfile. This new solution could both improve performance and reduce the effort required for program development.

This simple example illustrates how extended addressability can both improve the performance of existing solutions and open the possibility of new solutions. The large amounts of virtual and processor storage now available to an application can allow totally new solutions and simplify the entire process of application development.

Basic concepts

No single technique for extended addressability meets all possible needs. Choosing the right one for a particular application requires you to understand the advantages and disadvantages of the technique and some of the key differences between them. Many applications require a combination of various techniques. Before you decide to incorporate one or more of the techniques in the design of a new application, or decide to use a technique to modify an existing application, consult the detailed technical description of each technique.

Asynchronous cross memory communication

Asynchronous cross memory communication is a fancy way to describe scheduling an SRB. An SRB is a service request block that a task can schedule to request that some service take place in the same address space or another address space. Any data that the requesting task and the service share must be placed in common storage.

SRBs are one way to overlap processing. A task schedules an SRB to perform a service, then continues with its work. When the service completes, it informs the task. The timing, however, is asynchronous; the point when the SRB completes cannot be predicted.

Technical description

See "Asynchronous Inter-Address Space Communication" in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Synchronous cross memory communication

Synchronous cross memory communication, called cross memory, is both more complex and more flexible than scheduling an SRB. Cross memory requires the programmer to use MVS macros to establish a cross memory environment. This environment clearly defines the authorization requirements that protect the integrity of the address spaces involved. Once this environment is established, the application can use assembler instructions to transfer control from one address space to another.

Cross memory applications (as well as applications running in a single address space) can use the processor-managed linkage stack to simplify program linkages. In a cross memory environment, the program call (PC) instruction that transfers control to another routine can be either a basic PC or a stacking PC. If it is a stacking PC, the system saves status on the linkage stack before it passes control to the PC routine. When the PC routine returns control, the system automatically restores status from the linkage stack.

The key fact to remember, however, is that cross memory provides synchronous communication or processing across address spaces. When a task issues a PC instruction, control passes to the PC routine. When the PC routine completes, it returns control to the calling routine. Cross memory, for example, allows an application running in one address space to provide services for many users in other address spaces.

Technical description

See Chapter 2, “Linkage stack,” on page 11 and Chapter 3, “Synchronous cross memory communication,” on page 19. Chapter 8, “Creating address spaces,” on page 197 contains related information.

Access register ASC mode

In access register address space control (ASC) mode, a program can use the full set of assembler instructions (except MVCP and MVCS) to manipulate data in another address space or in a data space. Unlike cross memory, access registers allow full access to data in many address spaces or data spaces.

ASC mode determines how the processor resolves address references for the executing program. In primary ASC mode, the processor uses the contents of general purpose registers to resolve an address to

a specific location. In access register ASC mode, an access register (AR) identifies the space the processor is to use to resolve an address. The processor uses the contents of an AR as well as the contents of general purpose registers to resolve an address to a specific location.

In AR ASC mode, a program can move, compare, or perform operations on data in other address spaces or in data spaces. It is important to understand, however, that ARs do not enable a program to transfer control from one address space to another. That is, you cannot use ARs to transfer control from a program in one address space to a program in another address space. For that, you need cross memory.

You can, however, use ARs without using cross memory. If your application needs to manipulate data in other address/data spaces but does not need to transfer control to other address spaces, use ARs. If your application needs to transfer control to routines in other address spaces but does not need to manipulate data, use cross memory. If your application needs both the transfer of control and the manipulation of data, use both cross memory and ARs.

Technical description

See Chapter 5, “Using access registers,” on page 93.

Data-in-Virtual

Data-in-virtual enables you to map data into virtual storage but deal only with the portion of it that you need. The DIV macro provides the system services that manage the data object. It enables you to map the object into virtual storage, create a window, and "view" through that window only the portion of the data object that you need. The system brings into central storage only the data that you actually reference.

You can map a data-in-virtual object in either an address space, a data space, or a hiperspace. Mapping the object into a data space or hiperspace provides additional storage for the data; the size of the window is no longer restricted to the space available in an address space. It also provides additional isolation and integrity for the data, as well as more direct methods of sharing access to that data.

Data-in-virtual is most useful for applications, such as graphics, that require large amounts of data but normally reference only small portions of that data at any given time. It requires that the source of the object be a VSAM linear data set on DASD (a permanent object) or a hiperspace (a temporary object).

Data-in-virtual is also useful for applications that require small amounts of data; data-in-virtual simplifies the way you access data by avoiding the complexities of access methods.

Technical description

See "Data-in-Virtual" in *z/OS MVS Programming: Assembler Services Guide*.

Virtual lookaside facility

The virtual lookaside facility (VLF) is a set of MVS services that provide a high-performance alternate path method of retrieving named objects from DASD on behalf of many users. VLF is designed primarily to improve the response time for such applications.

VLF uses data spaces to hold data objects in virtual storage as an alternative to repeatedly retrieving the data from DASD. If you have an existing data retrieval application or are considering designing one, determine whether VLF can meet your needs.

Technical description

See "Virtual Lookaside Facility (VLF)" in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Data spaces and hiperspaces

Data spaces and hiperspaces are data-only spaces that can hold up to 2 gigabytes of data. They provide integrity and isolation for the data they contain in much the same way as address spaces provide integrity and isolation for the code and data they contain. They are an extremely flexible solution to problems

related to accessing large amounts of data. There are two basic ways to place data in a data space or a hiperspace. One way 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. For hiperspaces, this second way reduces the amount of I/O.

Programs that use data spaces run in AR ASC mode. They use MVS macros to create, control, and delete data spaces. Assembler instructions executing in the address space directly manipulate data that resides in data spaces.

Programs that use hiperspaces run in primary or AR ASC mode. They use MVS macros to create, control, and delete hiperspaces. Programs cannot directly manipulate data in a hiperspace, but use MVS macros to transfer data to and from the hiperspace for data manipulation. Hiperspaces provide high-speed access to large amounts of data.

Technical description

To decide whether to use a data space or a hiperspace, see "Basic decision: data space or hiperspace" on page 5. More detailed information appears in Chapter 6, "Creating and using data spaces," on page 127 and Chapter 7, "Creating and using hiperspaces," on page 159.

Basic decision: data space or hiperspace

For storing data, MVS offers a program a choice of two kinds of virtual storage areas outside the program's address space: data spaces and hiperspaces. You must make these decisions:

- Does my program need virtual storage outside the address space?
- Which kind of virtual storage is appropriate for my program?

Data spaces and hiperspaces are similar in that both are areas of virtual storage that the program can ask the system to create. They differ in the way your program accesses data in the two areas. This difference, and others, are described in other sections. But before you can understand the differences, you need to understand what your program can do with these virtual storage areas.

Under certain conditions, virtual input/output (VIO) can be a better option than a data space or a hiperspace. "Choosing VIO instead of a data space or a hiperspace" on page 9 compares data spaces, hiperspaces, and VIO, and presents some trade-offs.

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. You can restrict access to data in those spaces to one or several units of work.
- Share data among programs that are executing in the same address space or different address spaces.

  Instead of keeping the shared data in common areas, create a data space or hiperspace for the data you want your programs to share. 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 in the appropriate sections in this document.

How Does a Program Obtain a Data Space or a Hiperspace? Data spaces and hiperspaces are created through the same system service: the DSPSERV macro. On this macro, you request either a data space or a hiperspace. You also specify some characteristics for the space, such as:

- Its size
- Its name
- Its storage key
- Its fetch protection attributes

The macro service allocates contiguous virtual storage of the size (up to two gigabytes) you specify.

Who Owns a Data Space or Hiperspace? Although programs create data spaces and hiperspaces, they do not own them. When a program creates a data space or hiperspace, the system assigns ownership to the TCB that represents the program or to the TCB that your program chooses as the owner.

When a TCB terminates, the system deletes any data spaces or hiperspaces that the TCB still owns. If you want the space to exist after the creating TCB terminates, assign the space to a TCB that will continue to be active beyond the termination of the creating TCB.

Can Problem State Programs Use Data Spaces and Hiperspaces? Problem state programs can create and use both data spaces and hiperspaces. Some types of data spaces and hiperspaces require that a program be supervisor state or have PSW key 0-7.

What are the differences?

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 1 on page 7 shows a program in AR mode using a 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.

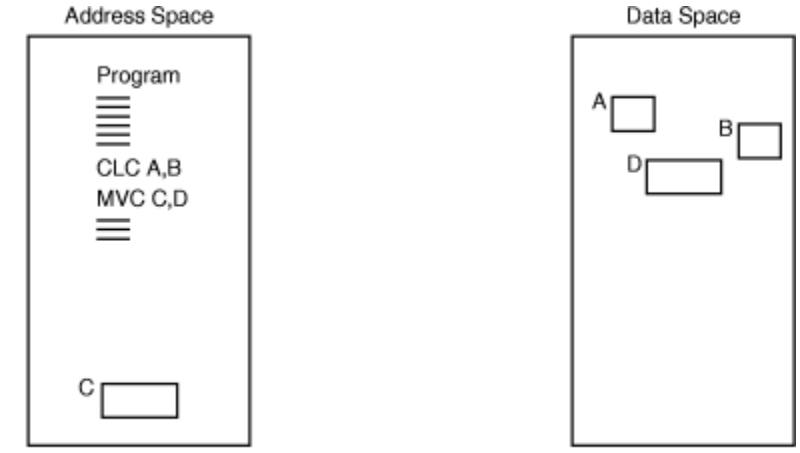


Figure 1. Accessing data in a data space

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 2 on page 7 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. While the data is in the address space, the program 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.

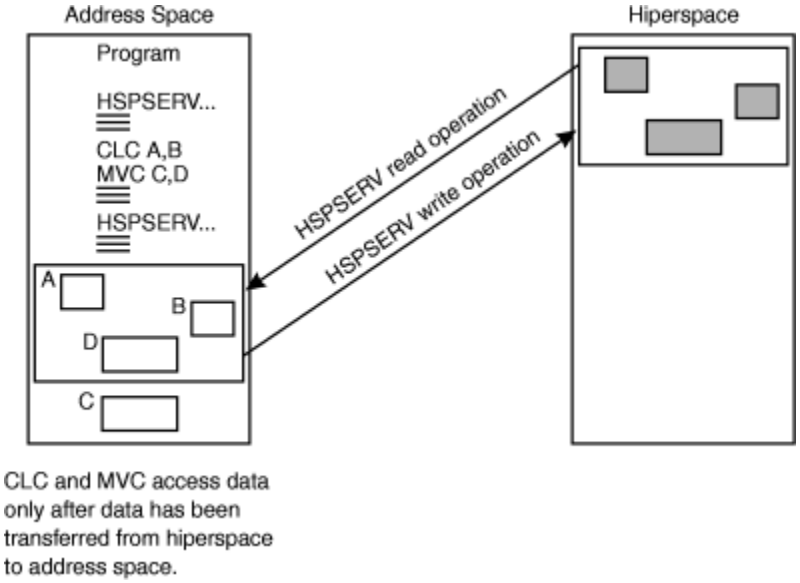


Figure 2. Accessing data in a hiperspace

With one HSPSERV invocation, the program can transfer 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 storage and expanded storage (if available) frames, and auxiliary storage slots. The system can move 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 only, or with a combination of expanded and auxiliary storage, depending on your choice. When you create a hiperspace, the system gives you storage that will not be the target of assembler instructions and will not need the backing of real storage frames. Therefore, when the system moves data from hiperspace to address space, it can make the best use of the available resources.

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 you frequently address 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 you need 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 ARs.
- The program can benefit from the high-speed access.
- The system can use the unused central 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.

Assume you want to change this application to improve its performance. 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 use address space buffers as an intermediate buffer area for transferring data to and from DASD.

Technical description

See Chapter 6, “Creating and using data spaces,” on page 127 for more information about data spaces.

An example of using a hiperspace

Suppose existing programs running in the same address spaces use 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 a program wants to reference a record, it reads the record in from DASD.

This kind of application can benefit from a hiperspace. If the data base were to exist in a hiperspace, a program would still bring one record into its address spaces 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.

Technical description

See Chapter 7, “Creating and using hiperspaces,” on page 159 for more information about hiperspaces.

Choosing VIO instead of a data space or a hiperspace

Virtual input/output (VIO), like data spaces and hiperspaces, is designed to reduce the need for the processor to transfer data between DASD and central storage. In this way, all three speed up the execution of your programs. Additionally, they all use expanded storage, where possible, to back the data. This section compares VIO with data spaces and hiperspaces and suggests the circumstances under which you would choose VIO.

In making the decision on which to chose, you need to consider the following questions:

- How is the data in your program organized?
- How does the program use the data?
- How much programming effort is required to change an existing program to take advantage of VIO, data spaces, or hiperspaces?

Two tables in this section help you understand facts related to these questions. Table 1 on page 9 answers questions about the data the program uses.

Table 1. Data requirements for VIO, data spaces, and hiperspaces

Question	VIO	Data spaces	Hiperspaces
Is the data in your program temporary?	VIO supports only temporary data.	Data spaces support temporary data and permanent data (through DIV).	Hiperspaces support temporary data and permanent data through DIV or data window services. (For information on using data window services, see *z/OS MVS Programming: Assembler Services Guide*.)
Is the data in your program sequential?	VIO (through EXCP) supports both sequential and random access; however, random access requires more processor cycles.	Data spaces have no requirement.	Hiperspaces have no requirement.
Is data arranged (or able to be organized) in 4K byte blocks?	VIO has no requirement.	Data spaces have no requirement.	Hiperspaces require that the data be accessed and referenced in 4K byte increments, located on a 4K byte boundary.

You might have an existing program — either an assembler program or a high-level-language (HLL) program — that you would like to change to use the performance benefits of VIO, data spaces, or hiperspaces. Table 2 on page 10 compares the programming effort required to make this change.

Table 2. Difficulty of modifying an existing application			
Question	**VIO**	**Data spaces**	**Hiperspaces**
How difficult is it to modify existing programs that use I/O operations?	VIO requires no modification to existing programs that use an access method that uses EXCP. Either use storage management subsystem (SMS) to make global requests to use VIO or use JCL for an individual program.	Assembler programs must change to use system macros and access registers. Through VLF, authorized programs can use data spaces to store and retrieve named objects. HLL programs cannot use data spaces directly.	Assembler programs must change to use system macros or data window services. HLL programs cannot use hiperspaces directly. They can use hiperspaces through data window services.

What is the performance comparison? Data spaces and hiperspaces do not have the overhead of an access method and the device simulation of VIO; therefore, they require less processor time than VIO.

When would you choose VIO over data spaces or hiperspaces? Use VIO when you want to improve the performance of an existing program, but you do not want to make large changes. For information about how to use VIO, see *z/OS MVS JCL User's Guide*.

Chapter 2. Linkage stack

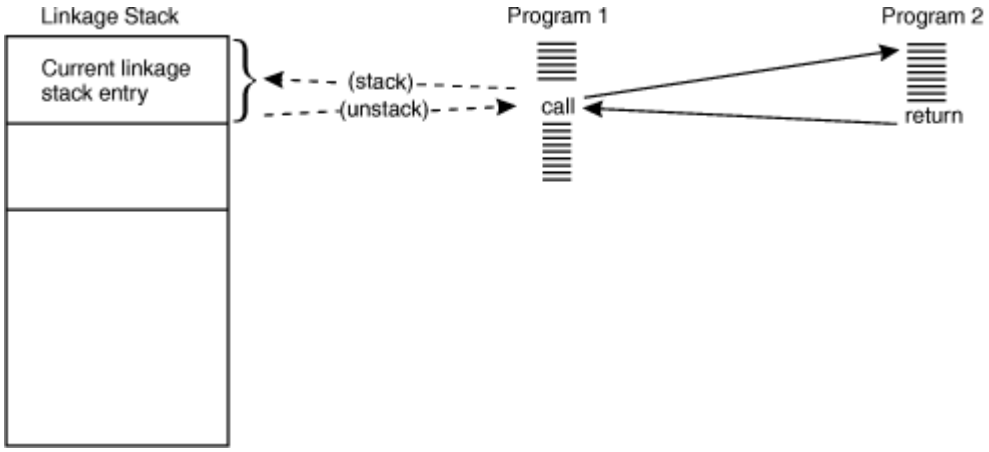
The **linkage stack** is an area of protected storage that the system gives a program to save status information at a branch or a program call. This section describes the linkage stack and the assembler instructions that cause the system to add and remove an entry and modify the entry.

Saving status is a required part of program linkage. Status includes general purpose registers (GPRs), access registers (ARs), the PSW, and other important information. The first thing a program does when it receives control is save the status of its caller. The last thing the program does before it returns control is restore the caller's status. The calling program can then resume processing with its status (including registers and cross memory environment) intact. For example, your PC routines might have used the PCLINK STACK macro to save a caller's status and then the PCLINK UNSTACK macro to restore the status.

An easier way to save and restore status, however, is to allow the system to do it for you through the linkage stack. The linkage stack saves you and the system work in the following ways:

- The "chain" of status save areas is located in one place rather than chained throughout storage. Diagnostic information thus appears in sequence on the linkage stack. (For the contents of an entry in the stack, see “Contents of the linkage stack entry” on page 14.)
- The linkage stack provides a place for reentrant programs to save the caller's complete status before the reentrant programs dynamically obtain their working storage. Once a program has saved the caller's status on its linkage stack, it has all 16 GPRs and ARs available to establish its working environment.
- Your programs do not have to obtain and chain 72-byte save areas, provided all called programs are using the linkage stack.

The following illustration shows how a program uses the linkage stack. The call from Program 1 to Program 2 automatically places all the caller's status on the linkage stack. The return from Program 2 to Program 1 automatically restores all the caller's status and removes the entry from the linkage stack.



The system provides each workunit (that is, TCB or SRB) with its own linkage stack. The linkage stack is then available to all programs that the workunit represents. The programs can run in primary or AR address space control (ASC) mode. They can be problem state or supervisor state, locked or unlocked, enabled or disabled.

The linkage stack actually consists of two stacks: the normal linkage stack and the recovery linkage stack. The **normal linkage stack** consists of at least 96 entries (for tasks) or 57 entries (for SRBs) for use by programs that run under the workunit. (Note that under some circumstances, the system might provide more than this.) When the system needs an entry and finds that all entries in the normal stack are used, it abends the program with a "stack full" interruption code. After the "stack full" interruption occurs, the system uses the recovery linkage stack. The **recovery linkage stack** is available to the program's recovery routines after the "stack full" interruption occurs. If you anticipate a need for more than 96 entries, you can use the LSEXPAND macro to expand the size of the normal and recovery stacks for tasks. For

information about how and when to issue the LSEXPAND macro, see "Expanding a linkage stack to a specified size" on page 16.

Linkage stack considerations for asynchronous exit routines

A user may request an asynchronous exit routine to execute on behalf of a task. When an asynchronous exit routine gets control, it cannot access the last entry (if any) on the linkage stack, because that entry was created by the interrupted routine. The extract stacked registers (EREG) instruction, extract stacked state (ESTA) instruction, and the modify stacked state (MSTA) instruction will cause a linkage stack exception to occur.

Any routines to which the exit routine passes control are also subject to the same restriction. However, the exit routine, or any routines to which it passes control, can manipulate linkage stack entries that they themselves add.

Instructions that add and remove a linkage stack entry

The three instructions that cause the system to add or remove entries on the linkage stack are:

- The stacking program call (PC), which adds an entry when it passes control to another routine.
- The branch and stack (BAKR), which adds an entry whether it branches to another routine or not.
- The program return (PR), which removes an entry when it returns from a call or branch made with either a stacking PC or a BAKR.

This section introduces each instruction and gives simple examples of each. It is not intended to direct you in your coding. For complete descriptions of the instructions, see *Principles of Operation*.

The stacking PC instruction adds an entry to the linkage stack. Chapter 3, "Synchronous cross memory communication," on page 19 describes the two types of PC linkages. The **stacking PC** uses the linkage stack to save the user's environment. The **basic PC**, on the other hand, requires that the PC routine provide code to save the user's environment. The stacking and basic PC instructions are cross memory instructions; they are described in more detail in "PC linkages" on page 23. Chapter 3, "Synchronous cross memory communication," on page 19 also contains a comparison of the coding of a stacking PC and a basic PC. The linkage stack instructions BAKR and PR are described in the following sections.

Branch and stack (BAKR) instruction

The branch and stack (BAKR) instruction performs the branch and link in the same way that the BALR does; additionally, it adds an entry to the linkage stack. The entry includes the branch address of the calling routine.

A program can use the BAKR to branch to a subroutine in its address space and add an entry to the linkage stack, or it can use a BAKR simply to branch to the next instruction in the program and add an entry to the linkage stack. A program return (PR) instruction returns control to the program and removes an entry from the linkage stack.

The BAKR instruction does not change the current addressing mode, nor does it cause a branch out of an address space. You can be in either primary or AR ASC mode to use BAKR.

Program return (PR) instruction

The PR instruction performs several actions on the **current entry** in the linkage stack — the current entry being the entry formed by the most recent BAKR or stacking PC instruction:

- If the current entry was added by a stacking PC or a BAKR instruction, the PR instruction returns control to the calling program.
- The PR instruction restores the contents of the current entry, including the cross memory environment, the PSW, and the contents of registers 2 - 14.
- The PR instruction removes the current linkage stack entry.

The PR instruction can execute in either primary or AR mode.

The following example shows the PR instruction and a use of the BAKR instruction.

CALLING PROGRAM	SUBROUTINE
. . L 15,=A(SUBR) BALR 14,15	. . . SUBR EQU * BAKR 14,0 . . PR

In the example, the BALR branches to subroutine SUBR. When SUBR receives control, it uses BAKR to save the caller's status on the linkage stack. The BAKR saves the contents of register 14, which the calling program loaded with the address of the instruction after the BALR, on the linkage stack. Zero, as the second operand, means that the status information is saved and no branch occurs. The PR instruction in SUBR restores the caller's status, restores the contents of register 14, removes the current linkage stack entry, and returns to the instruction after the BALR in the calling program.

This use of the BAKR instruction is consistent with the MVS convention in which the called program saves the status of the caller. This convention is described in the section on linkage conventions in *z/OS MVS Programming: Assembler Services Guide*.

Example of using the linkage stack

Figure 3 on page 13 shows how the stacking PC and the BAKR instructions add entries to TCBA's linkage stack and how the PR instruction removes those entries.

The program call from Program 1 to Program 2 automatically places all the caller's status on the linkage stack (adding Entry 1 to the linkage stack). Program 2 uses the BALR instruction to branch to a subroutine, which uses the BAKR instruction to save Program 2's status (adding Entry 2 to the stack). When the subroutine returns to Program 2 through the PR instruction, Program 2's status is restored (removing Entry 2 from the stack). When Program 2 uses the PR instruction to return to Program 1, Program 1's status is restored (removing Entry 1 from the stack). At any time, the entry formed by the most recent BAKR or stacking PC instruction contains the status of the caller of the currently executing code.

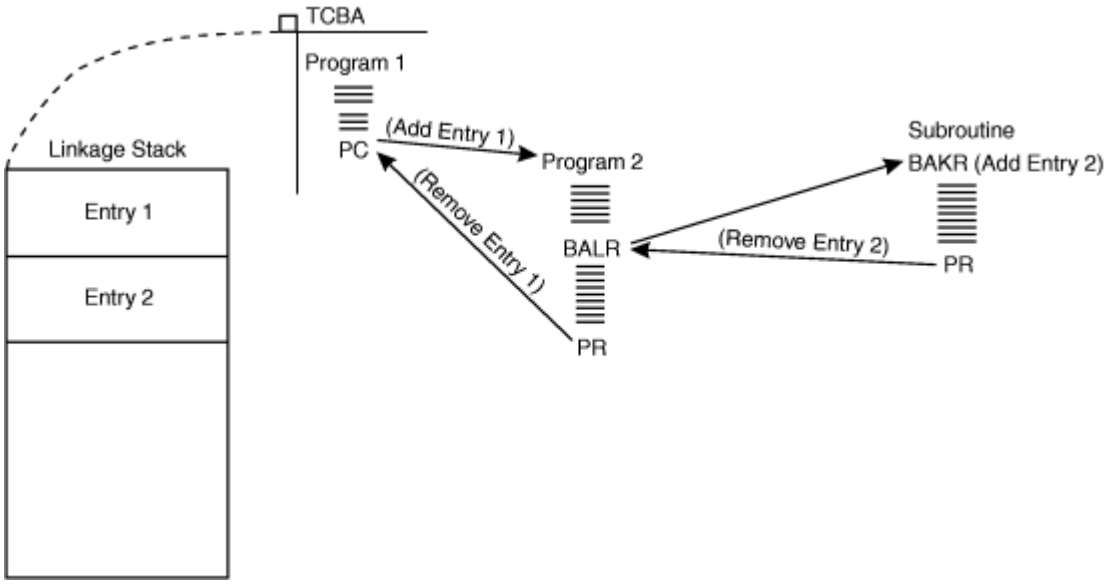


Figure 3. Example of using the linkage stack

Contents of the linkage stack entry

A **linkage stack entry** includes the following information:

- Contents of GPRs 0 - 15
- Contents of ARs 0 - 15
- Primary and secondary address space numbers (PASN and SASN)
- EAX
- Entire current PSW
- PSW key mask (PKM)
- PC number (if a stacking PC caused the entry) or a branch address (if a BAKR caused the entry)
- An eight-byte area that you can change with the modify stacked state (MSTA) instruction

On return from a routine, the PR instruction restores the entry and returns control to the calling program.

Instructions that manipulate the contents of a linkage stack entry

A program cannot change the order of the entries on the linkage stack, nor can it change any part of an entry, except for the eight-byte modifiable area of the current entry. Three instructions copy information from the current entry or copy information to the modifiable area of the current entry:

- The extract stacked registers (EREG) instruction loads ARs and GPRs from the current linkage stack entry.
- The extract stacked state (ESTA) instruction obtains non-register information from the current linkage stack entry.
- The modify stacked state (MSTA) instruction copies the contents of an even/odd GPR pair to the modifiable area of the current linkage stack entry.

These instructions can execute in both primary and AR ASC mode.

Extract stacked registers (EREG) instruction

Use the extract stacked registers (EREG) instruction to load ARs and GPRs from the current linkage stack entry. A typical use of EREG is in the middle of a subroutine after the caller's input registers have been modified for other purposes. Use EREG to restore the contents of the AR/GPR pairs.

In the following example, EREG extracts the contents of the calling program's ARs 0-1 and GPRs 0-1 from the current entry and loads them into ARs 0-1 and GPRs 0-1. The entry in this example was caused by the BAKR instruction.

Table 3. Example of an EREG instruction.

The table shows how EREG extracts contents from the calling program and loads them.

CALLING PROGRAM	SUBROUTINE

`.`
`L    15,=A(SUBR)`
`BALR 14,15`
`.`
`.`
`.`
`.`
	`SUBR EQU  *`
	`     BAKR 14,0`
	`     .`
	`     .`
	`     .`
	`     EREG 0,1`
	`     .`
	`     .`
	`     PR`

The EREG instruction does not change the current stack pointer.

Another example of using EREG to extract the contents of an AR/GPR pair appears in “Example of using TESTART” on page 121.

Extract stacked state (ESTA) instruction

A linkage stack entry includes the contents of the ARs and the GPRs, as well as other information. The EREG instruction copies the ARs and GPRs from the current entry; the ESTA instruction copies the rest of the information (that is, the non-register information) from the current entry. The non-register information is divided into four eight-byte **information fields** and is identified to the ESTA instruction by a code.

Figure 4 on page 15 shows the code for each of the four information fields and the format of the fields. The format of the third field varies depending on whether a BAKR or a stacking PC instruction caused the entry.

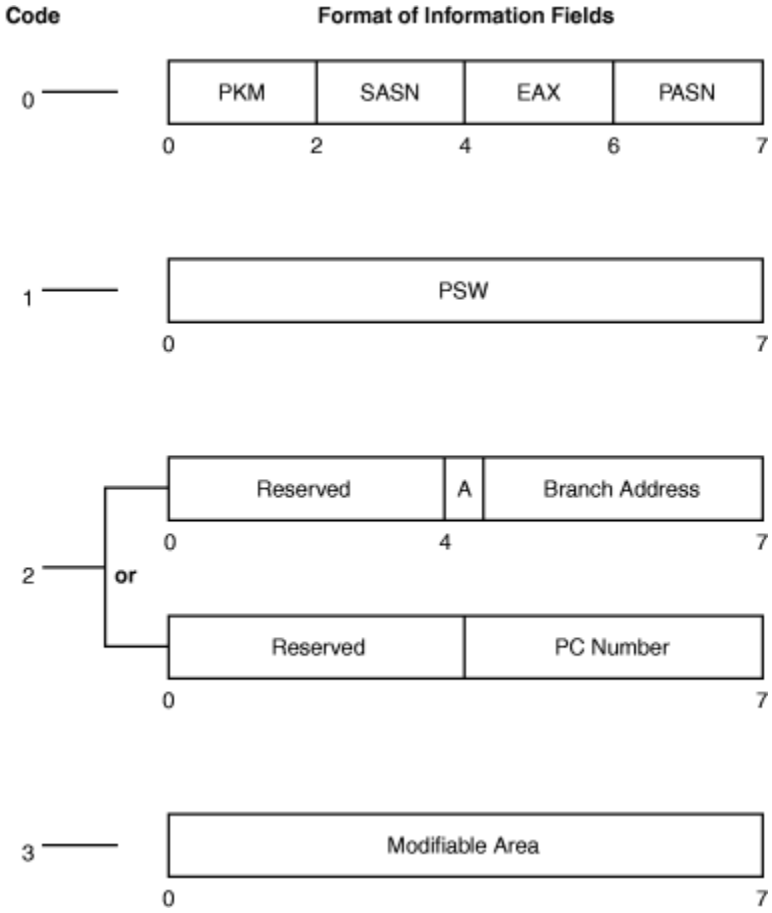


Figure 4. Format of the information fields

The ESTA instruction copies one of the fields in the current entry into an even/odd pair of GPRs. It returns a condition code that tells whether the entry on the linkage stack was formed by the BAKR (CC=0) or stacking PC (CC=1) instruction.

In the following example, the load address instruction (LA) loads a code of "1" into register 9, where "1" identifies the information field that contains the PSW. The ESTA instruction then copies this field into general registers 4 and 5. The BZ instruction causes a branch if the stack entry was formed by a BAKR.

```
* Program entered through a stacking PC or BAKR
* Code of 1 identifies the PSW in the linkage stack entry
    .
    .
    LA   9,1         Load the code of 1 into general register 9
    ESTA 4,9         Load the PSW into general registers 4 and 5
    BZ   BAKRTYPE    If CC=0, then BAKR formed the stack entry
                     If CC=1, then stacking PC formed the entry
```

Figure 5. Example of an ESTA instruction

Another example of the ESTA instruction appears in Figure 39 on page 121.

Modify stacked state (MSTA) instruction

The MSTA instruction moves the contents of an even/odd pair of GPRs into the modifiable area of the current linkage stack entry. You might use the ESTA instruction later to load the contents of the modifiable area into registers.

In the following example, general registers 6 and 7 contain 8 bytes to be placed into the modifiable area of the current linkage stack entry. The MSTA instruction copies the 8 bytes to the modifiable area. The load address (LA) instruction loads a code of 3 into GPR 1. (The code for the modifiable area is "3".) Later in the example, the ESTA instruction copies the contents of the modifiable area into general register 2.

```
* Program entered through a stacking PC or BAKR
* General registers 6 and 7 contain 8 bytes to be placed
* into modifiable area of the current linkage stack entry
* Code of 3 identifies the modifiable area in entry
    .
    .
    MSTA 6        Update modifiable area
    .
    .
    LA   1,3      Load code of 3 into general register 1
    ESTA 2,1      Load modifiable area into general registers 2 and 3
```

Figure 6. Example of an MSTA instruction

For an example of using MSTA to pass data to an associated recovery routine (ARR), see the section on providing recovery in the *z/OS MVS Programming: Authorized Assembler Services Guide*.

Expanding a linkage stack to a specified size

The system provides a way for programs running in task mode to expand the size of the normal and recovery linkage stacks. The default size for a normal linkage stack is at least 96 entries. A linkage stack of this size is probably sufficient for your program's needs. However, if you have a program, such as one with recursive code, that needs more than 96 entries, you can use the LSEXPAND macro to request a normal linkage stack of up to 16,000 entries.

When a program uses up all of the entries in the normal linkage stack, the system abends the workunit and sets up a recovery linkage stack. The default for this linkage stack is 24 entries. The LSEXPAND macro can increase the recovery linkage stack to 4000 entries.

The timing of the execution of the LSEXPAND macro is important. You must anticipate using up the entries in the stack. If the program has already issued a "stack full" program interruption, the system will not accept the LSEXPAND macro and will abend the workunit. In other words, don't wait until the normal linkage stack is full to issue this macro.

Example of requesting larger normal and recovery linkage stacks

To request that the linkage stack have 2000 entries and the recovery linkage stack have 150 entries, code the following LSEXPAND macro:

```
LSEXPAND NORMAL=2000,RECOVERY=150
```

Relationship between the linkage stack and ESTAE-type recovery routines

When a user provides an ESTAE-type recovery routine through the ESTAE or ESTAEX macro, the system saves the current linkage stack position. The user must deactivate the ESTAE-type recovery routine under the linkage stack entry that was current when the ESTAE-type recovery routine was activated. *z/OS MVS*

Programming: Authorized Assembler Services Guide describes how the macros that provide recovery for your programs use the linkage stack.

Dumping the contents of the linkage stack

In case of an error, you might want to check the status information that the system saved when your program gained control. Through the interactive problem control system (IPCS) FORMAT line command, you can display or print dump data associated with a specified address space. *z/OS MVS Diagnosis: Tools and Service Aids* contains an example of a dump of an entry in the linkage stack.

Chapter 3. Synchronous cross memory communication

Synchronous cross memory communication enables one program to provide services synchronously to other programs. This document calls the program that provides the services, *the service provider*, and the programs that use the services, *users*. The service provider can provide services to one user or to many.

Synchronous cross memory communication takes place between the user and the service provider when the user issues a program call (PC) instruction. If the service provider has previously established the necessary environment, the PC instruction transfers control to a service provider program called a *PC routine*. The PC routine provides the requested service and then returns control to the user.

The user program and the PC routine can execute in the same address space or in different address spaces. In either case, the PC routine executes under the same TCB as the user. Thus, the PC routine provides the service synchronously.

A PC routine can access (fetch or store) data in the user's address space by using access registers (ARs) and the full set of assembler instructions. If the PC routine has the proper authority, it can also access data in other address spaces or in data spaces. For information about using access registers, see Chapter 5, “Using access registers,” on page 93.

The rest of this section discusses when you should consider using synchronous cross memory communication and explains the environment the service provider must create and how to create it.

When should you use synchronous cross memory communication?

The use of synchronous cross memory communication to provide services to users can provide virtual storage constraint relief as well as improve the integrity of the service and its data. Consider using synchronous cross memory communication if you wish to:

- Isolate the service and its data from the user of the service
- Make the service available to multiple users without the need to store it in commonly addressable storage
- Replace an existing service request block (SRB) routine to gain improved performance or simplify communication
- Provide an authorized service to problem state programs

Synchronous cross memory communication enables you to provide services to many users without making the service available in commonly addressable storage. At the same time, you can isolate the service from the user, thus protecting it by having the service in its own address space.

Terminology, macros, and assembler instructions

To fully understand the rest of the cross memory discussion, there are some terms that you must understand and macros and assembler instructions that you must become familiar with.

Cross memory terminology

The following terms are used within this section. For definitions of other terms used in this document, see the glossary.

- **Address space control (ASC) mode:** The mode (determined by the PSW) that tells the system where to find the data it is to reference. Two ASC modes are AR and primary. For each ASC mode, the following table defines:
  - The address space from which the system fetches instructions

  - The address space or data space that the system accesses when an instruction, other than an MVCP or MVCS instruction, references data
  - The address space the system accesses when an MVCP or MVCS instruction references data
- **AR ASC mode:** The ASC mode in which the system uses both the GPR (used as the base register) and the corresponding AR to resolve an address in an address/data space.

ASC Mode	Instruction Fetch	Data Access	MVCP/MVCS
Primary ASC mode	Primary address space	Primary address space	Primary or secondary address space
Secondary ASC mode	Primary address space (see Note)	Secondary address space	Primary or secondary address space
AR ASC mode	Primary address space	Address space indicated by the AR	Unavailable. Causes an abend in AR ASC mode.
Note: Prior to ESA/370 architecture, the address space from which instructions are fetched is unpredictable when a program is running in secondary ASC mode.			

- **Basic PC:** Transfers control to another program, the PC routine. The basic PC requires the service provider to save and restore the user's environment. The PC routine can be in the same address space as the program that issues the PC instruction, or in a different address space.
- **Cross memory local (CML) lock:** The LOCAL lock of an address space other than the home address space.
- **Cross memory mode:** Cross memory mode exists when at least one of the following conditions are true:
  - The primary address space (PASN) and the home address space (HASN) are different address spaces.
  - The secondary address space (SASN) and the home address space (HASN) are different address spaces.
  - The ASC mode is secondary.
- **Home address space:** The address space in which MVS initially dispatches a TCB or SRB (work unit). In the case of a TCB, the home address space contains the TCB. PSAAOLD points to the home address space. When MVS initially dispatches a work unit, the home address space, the primary address space, and the secondary address space are all the same. During execution of the work unit, the home address space remains the same. The primary and secondary address spaces may be changed, however, through the PC, PR, PT, or SSAR instructions.
- **LX reuse facility:** The LX reuse facility is available with z/OS V1R6 to provide additional LXs and improve reusability of LXs. It is enabled when running on a z890 or z990 processor at driver level 55 or above, with APAR OA07708 installed. When the facility is enabled bit CVTALR in byte CVTFLAG2 of the CVT data area is 1.
- **Primary address space:** The address space whose segment table is used to fetch instructions in primary, secondary, and AR ASC modes. A program in primary mode fetches data from the primary address space.
- **Primary ASC mode:** The ASC mode in which the system uses the GPRs, but not the ARs, to resolve an address in an address space. In primary ASC mode, the system fetches instructions and data from the primary address space.
- **PC number:** A number that identifies a PC routine. The service provider creates the number, by using MVS services, and supplies it to the user. The user specifies the number in a PC instruction to identify the PC routine that is to be invoked.
- **PC routine:** A program that receives control as the result of a PC instruction's executing and performs a service for the caller.

- **Secondary address space:** The address space whose segment table the system uses to access data in secondary ASC mode.
- **Secondary ASC mode:** The ASC mode in which the system fetches instructions from the primary address space and data from the secondary address space.
- **Space switch routine:** A program that issues a PC instruction that causes the primary address space to change.
- **Stacking PC:** Transfers control to another program, the PC routine. The stacking PC uses the linkage stack for storing the caller's status. It provides more options and more automatic function than the basic PC instruction. The PC routine can be in the same address space as the program that issues the PC instruction, or a different address space. **IBM recommends** using the stacking PC instead of the basic PC.

Macros used for synchronous cross memory communication

MVS provides the following macros that the service provider uses to create, disconnect, or destroy the environment (tables, linkages, and indexes) needed for cross memory communication. This section discusses when and how to use these macros. For detailed information about the macro's syntax and parameters, see one of the following:

- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

A brief description of each macro follows.

- **ATSET** (Set Authority Table): Sets PT and SSAR authority in the home address space's authority table entry that corresponds to a specified authorization index.
- **AXEXT** (Extract Authorization Index): Returns the authorization index (AX) value of a specified address space.
- **AXFRE** (Free Authorization Index): Frees one or more authorization index (AX) values by returning them to the system.
- **AXRES** (Reserve Authorization Index): Reserves one or more authorization index (AX) values for use by the caller of the macro.
- **AXSET** (Set Authorization Index): Sets the authorization index (AX) of the home address space to the value specified by the caller of the macro.
- **ETCON** (Connect Entry Table): Connects one or more entry tables to specified linkage table indexes in the home address space.
- **ETCRE** (Create Entry Table): Builds a program call (PC) entry table from PC routine definitions that the service provider defined by issuing the ETDEF macro or by defining the entry definitions directly.
- **ETDEF** (Create Entry Table Descriptor): Builds or modifies the PC routine definitions that ETCRE uses as input.
- **ETDES** (Destroy Entry Table): Destroys an entry table.
- **ETDIS** (Disconnect Entry Table): Disconnects one or more entry tables from the home address space's linkage table.
- **LXFRE** (Free a Linkage Index): Frees one or more previously reserved linkage indexes.
- **LXRES** (Reserve a Linkage Index): Reserve one or more linkage indexes for future use.
- **PCLINK** (Stack, Unstack, or Extract Program Call Linkage Information): Basic PC routines that receive control in supervisor state can issue
  - Save the user's environment after the PC routine gets control.
  - Restore the user's environment before issuing the PT instruction to return control to the user.
  - Retrieve information from a saved environment.

Instructions used for cross memory communication

The following assembler instructions provide the control capabilities and information a program needs for synchronous cross memory communication. For a detailed explanation of the following instructions, see *Principles of Operation*.

- **EPAR** (extract primary ASN) - Places the ASID of the primary address space into a general purpose register (GPR).
- **EPAIR** (extract primary ASN and instance) - Places the ASID and instance number of the primary address space into a general purpose register (GPR). Bits 0-31 of the GPR contain the instance number and bits 48-63 contain the ASID.
- **ESAR** (extract secondary ASN) - Places the ASID of the secondary address space into a general purpose register.
- **ESAIR** (extract secondary ASN and instance) - Places the ASID and instance number of the secondary address space into a general purpose register (GPR). Bits 0-31 of the GPR contain the instance number and bits 48-63 contain the ASID.
- **IAC** (insert address space control) - Indicates, in a general purpose register, the current ASC mode.
- **MVCK** (move with key) - Moves data between storage areas that have different storage protection keys.
- **MVCP** (move to primary) - Moves data from the secondary address space to the primary address space.
- **MVCS** (move to secondary) - Moves data from the primary address space to the secondary address space.
- **PC** (program call) - Invokes the program identified by the specified PC number. There are two types of PC linkages, basic and stacking. Both linkages transfer control to another program, the PC routine. The stacking PC, however, provides more capability and better performance than the basic linkage. The PC routine that receives control can be in an address space other than the address space in which the PC instruction was issued.
- **PR** (program return) - Returns control to a program that issued a stacking PC instruction.
- **PT** (program transfer) - Returns control to a program that issued a basic PC.
- **PTI** (program transfer with instance) - Return control to a program that issued a basic PC.
- **SAC** (set address space control) - Explicitly sets the ASC mode.
- **SSAR** (set secondary ASN) - Sets the secondary address space to the desired address space.
- **SSAIR** (set secondary ASN and instance) - Sets the secondary address space and instance number to the desired address space and instance number.

An overview of cross memory communication

Cross memory communication takes place when a user program issues a PC instruction that specifies a valid PC number. The PC number identifies the PC routine that the system is to invoke. The service provider must have previously defined the PC routine and made the PC number available to the user. The PC routine can provide the requested service or can invoke other programs to provide the service. When the PC routine completes its function, it issues either the PR instruction (for a stacking PC) or a PT instruction (for a basic PC) to return control to the user program.

PC routines

To provide a service, the service provider supplies a PC routine and uses MVS macros to make the PC routine available to the address space of the user who needs the service. A service provider can make a PC routine available to users in all address spaces or to users in selected address spaces only. Before a user can invoke a PC routine, however, the user also needs a PC number, which the service provider must provide. Each PC number identifies a specific PC routine.

PC routine invocation

Any program can issue a PC instruction provided the program is running in either primary or AR ASC mode. When a program issues a PC instruction, the system invokes the requested PC routine providing the service provider has made the PC routine available to the calling program's address space. In addition, if the calling program is running in problem state, it must also have a PSW-key mask (PKM) that the service provider has authorized to invoke the PC routine. The PC routine, or other routines that it invokes, performs the service the caller desires.

The service provider is responsible for defining the level of authorization problem programs need to invoke a PC routine. The service provider defines the level of authorization for each PC routine by specifying the PKM that a problem state program must have to invoke the PC routine. If the problem program's PKM agrees with the service provider's specification, the system allows the problem program to invoke the PC routine. Otherwise, the PC instruction causes a program interrupt (privileged operation exception).

PC linkages

There are two types of PC linkages, *basic* and *stacking*. The terms basic and stacking refer to the type of PC linkage used to invoke the PC routines. The basic PC linkage and the stacking PC linkage are similar in that each invokes a program. They are different, however, in the manner of invocation and the capabilities available on invocation.

The stacking PC provides more capability and better performance than does the basic PC. For example, the stacking PC uses the system provided linkage stack to save and restore the user's environment. On the other hand, the basic PC requires that the PC routine provide code to save and restore the user's environment. For a detailed comparison of the stacking PC and the basic PC, see “PC linkages and PC routine characteristics” on page 45.

PC routine execution

A PC routine executes in either the service provider's address space or in the user's address space. A PC routine that causes a space switch (PC-ss) executes in the service provider's address space, which becomes the primary address space. A PC routine that does not cause a space switch (PC-cp) executes in the user's primary address space, which remains the primary address space. The service provider uses the ETDEF macro to indicate whether a PC routine is to cause a space switch. Regardless of where the PC routine executes, it always runs under the same TCB or SRB as the program that issued the PC instruction.

A PC routine may provide services directly to the user, or it may invoke other routines to provide the services. If necessary, a PC routine can itself issue PC instructions.

Stacking pc routines

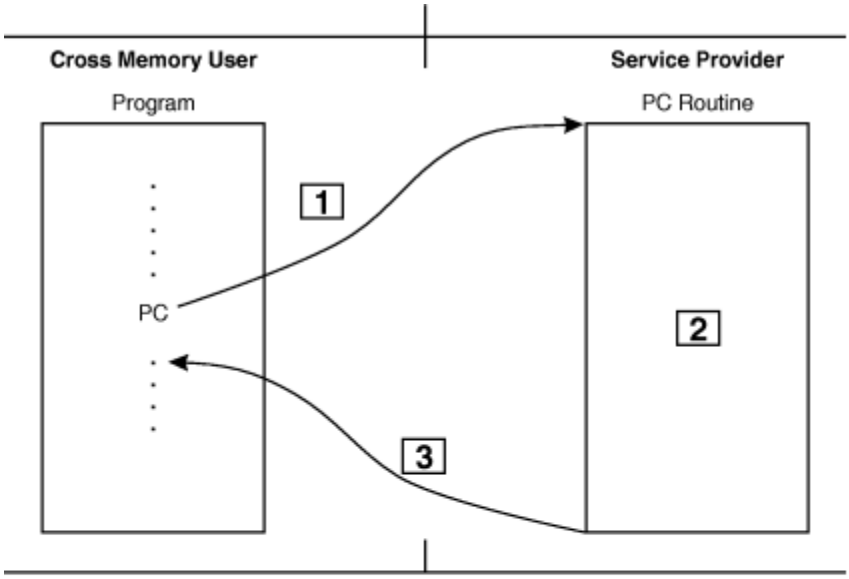
When a user invokes a stacking PC routine, the system saves the user's environment on the linkage stack. After the PC routine completes its function, it must issue the PR instruction to restore the user's environment and return control to the user.

Basic pc routines

When a user invokes a basic PC routine, the PC routine must save the user's environment. To save the environment, the PC routine can issue the PCLINK macro or can provide code that performs the save function. After completing its function, the PC routine must restore the user's environment. Again, the PC routine can use the PCLINK macro or provide code that performs the restore function. Only PC routines that run in supervisor state can issue the PCLINK macro. PC routines that run in problem state must provide code that performs the same functions as the PCLINK macro.

To return control to the user, the PC routine issues the PT instruction. To execute a PT instruction, the service provider must have previously been granted PT authority by the user. PT authority means that the service provider is authorized to issue the PT instruction with the target being the user's address space. The AXSET macro provides the means to grant PT authority.

The following figure shows how a user invokes a PC routine to obtain a service.



Example 1

1. A program in the cross memory user's address space issues a PC instruction to request a service.

   The system transfers control to the PC routine in the service provider's address space.

2. The PC routine in the service provider's address space performs the service.

3. After performing the service, a stacking PC issues a PR instruction and a basic PC issues a PT instruction to return control to the cross memory user's program.

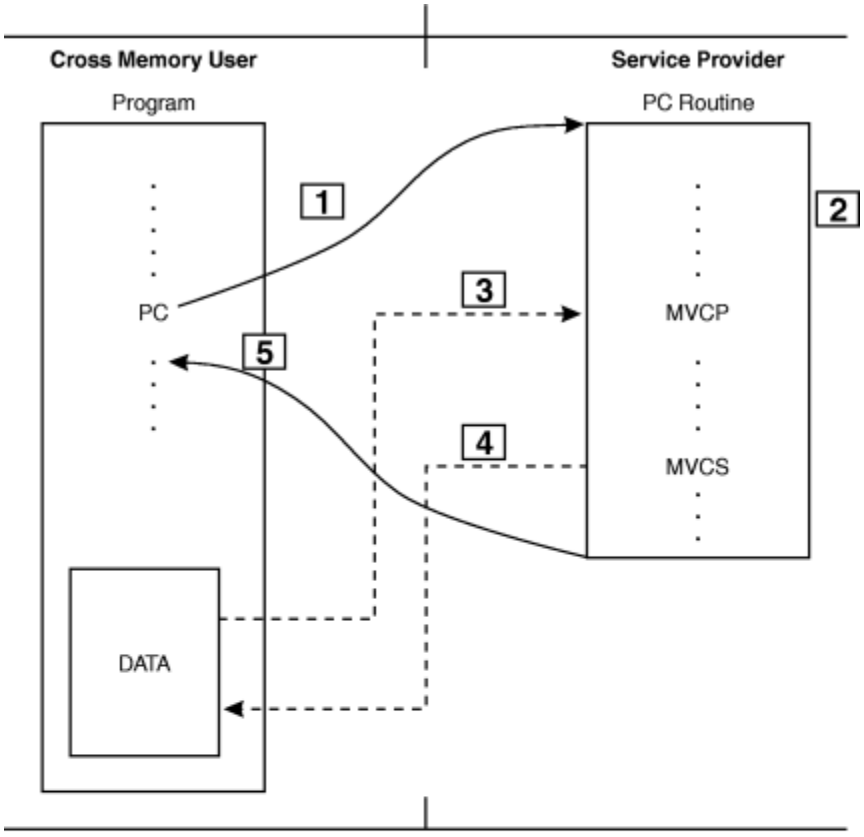
Figure 7. PC routine invocation

Accessing data from a pc routine

To access data that is in another address space or in a data space, or to store data into another address space or data space, **IBM recommends** that the PC routine use ARs. To use ARs, the PC routine must be in AR ASC mode. A PC routine can use ARs in the same way any other program uses them. For information about using ARs and for some examples, see Chapter 5, “Using access registers,” on page 93.

The PC routine can, if necessary, access data in the user's address space without using ARs. The MVCP instruction moves data from the secondary address space (the user) to the primary address space (the service provider). The MVCS instruction moves data from the primary address space (the service provider) to the secondary address space (the user). To use the MVCP or MVCS instructions, the service provider must have obtained SSAR authority to the user's address space before the PC routine receives control. The AXSET macro provides the means to grant SSAR authority. The primary address space and the secondary address space must be different address spaces (PASN¬=SASN).

The following figure shows how a PC routine can use the MVCP or MVCS instructions to access data.



Example 2

1. A program in the cross memory user's address space issues a PC instruction to request a service.

   The system transfers control to the PC routine in the service provider's address space.

2. The PC routine in the service provider's address space performs the service.

3. The PC routine retrieves data from the cross memory user.

4. The PC routine stores data back into the cross memory user's address space.

5. After performing the service the PC routine returns control to the cross memory user's program.

Figure 8. Accessing data through the MVCP and MVCS instructions

Summary of cross memory communication

There are several important points to remember about cross memory communication:

1. Cross memory facilities enable the service provider to provide services to some or all users.
2. The service provider code and the user code can execute in the same address space or in different address spaces.
3. The service provider uses MVS macros to establish and maintain the environment needed for cross memory communications.
4. The service provider supplies services through PC routines. For each PC routine, the service provider supplies the user with a PC number that identifies the routine.

5. To obtain a service from the service provider, the user issues a PC instruction. The instruction specifies the PC number of the PC routine that the user wants to invoke.
6. The stacking PC provides more capability and better performance than does the basic PC. **IBM recommends** using the stacking PC.
7. To store data into or retrieve data from other address spaces or from data spaces, **IBM recommends** using ARs. The service provider can, if necessary, access data in the user's address space without using ARs. To do this, the service provider can use the MVCP instruction to retrieve data from the user's address space and the MVCS instruction to move data into the user's address space.

The cross memory environment

The term cross memory environment refers to the tables and linkages that connect the service provider's address space to the user's address space and to the tables and linkages that provide the necessary authorization for the service provider. The term also refers to the PC numbers used to initiate cross memory communication.

Following this topic are two figures, Figure 9 on page 29 and Figure 10 on page 30, that show how the cross memory environment supports communication. Refer to these figures as you read this topic.

Entry tables

For each PC routine, the service provider issues the ETDEF macro to define the PC routine's name or entry point and its environment. A PC routine's environment refers to whether the routine runs in supervisor state or problem state, the value of the routine's authorization key mask, whether the routine causes a space switch, and so forth. After defining the PC routine's environment, the service provider issues ETCRE to create an entry table. The service provider's home address space owns the entry table.

The entry table contains one entry for each PC routine. Each entry contains the operating environment definition created by ETDEF. Before a user can invoke a PC routine, the service provider must connect the entry table to the linkage table of the user's address space. The system might place additional entries into the entry table, after those defined by the service provider. These additional entries help to ensure that use of a PC number that is not one of those defined by the service provider results in system completion code ‘0D6’.

Linkage tables

When the LX reuse facility is enabled, each address space has a linkage first table and a linkage second table. The linkage index locates a specific entry in the linkage first table that in turn is used to locate a specific entry in the linkage second table. Otherwise, each address space has a linkage table. The linkage index locates a specific entry in the linkage table. For simplicity, subsequent discussion and diagrams will not make this distinction and will refer to the linkage table.

There are two types of LXs, a non-system LX and a system LX.

non-system LX
Use to connect an entry table to the linkage table in one or more, but not all address spaces.

system LX
Use to connect an entry table to all linkage tables.

A system LX, for example, enables an installation to replace an installation written SVC routine with a PC routine that gets invoked through a system linkage index.

The PC number

The service provider must also supply the user with a PC number. The service provider creates this number by concatenating the LX to the entry table index (EX). As previously stated, the LX is an index into the linkage table. The EX is an index into the entry table and identifies the relative entry in the entry table that corresponds to the PC routine that is to receive control. **Example:** If the first table entry corresponded to the PC routine, the EX would be X‘00’; if it was the second entry, the EX would be X‘01’,

and so forth. The service provider is responsible for calculating and keeping track of entry table indexes. When a program issues the PC instruction, the system uses the PC number to locate the correct entry table entry and transfer control to the PC routine.

The service provider and the user must agree on a method the service provider will use to provide the user with the PC number. The service provider might, for example, supply a macro that returns the PC number to the user. Or the service provider could place the PC number in a storage area common to both the service provider and the user. The user could then retrieve the PC number from the common area.

In addition, when you use a reusable LX, the service provider and the user must also agree on a method the service provider will use to provide the user with the LX sequence number. The service provider could, for example, supply a macro that returns the PC number/LX sequence number to the user. Or the service provider could place the PC number/LX sequence number in a storage area that is common to both the service provider and the user. The user would then retrieve the PC and LX sequence number from the common area.

Program authorization — the PSW-key mask (PKM)

Each program has associated with it a *PSW-key mask* (PKM) value. The PKM is a string of 16 bits that represents storage protection keys that are valid for a problem state program to use, where bit n equal to 1 indicates that the program is authorized to use key n. The system uses the PKM to check the authorization of problem state programs only. Supervisor state programs do not require PKM authority.

For a problem state program, the PKM defines:

- The PSW key values that the program can set by means of the SPKA instruction
- The storage key values the program can specify on the MVCK, MVCS, and MVCP instructions
- PC routines that the program is authorized to invoke

All programs are initially dispatched with a PKM value representing the storage protect key of the program's TCB or SRB, plus key 9. Each bit in the 16-bit PKM corresponds to a specific key. For instance, a PKM value of `X'0080'` (bit 8, the 9th bit, is on) represents key 8 and `X'0001'` (bit 15, the 16th bit, is on) represents key 15. Thus, a key 8 program is initially dispatched with a PKM value of `X'00C0'` (having the bits representing key 8 and key 9 on). The PC, PR, and PT instructions can change the PKM value.

The entry that defines a PC routine in the entry table contains two fields that are related to the PKM. Those fields are the *authorization key mask* (AKM) and the *entry key mask* (EKM). The AKM is a 16-bit string value that indicates the keys that will authorize a problem state program to invoke the PC routine. A problem state program can invoke the PC routine if at least one bit in the PKM and the corresponding bit in the AKM are both on (that is, set to `B'1'`).

The EKM is a 16-bit string value like the PKM. It can be used to alter the PSW keys under which the PC routine will run. For a basic PC routine, the system ORs the EKM into the PKM before the PC routine receives control. The result of the OR operation is the PKM under which the PC routine will run. A stacking PC routine can either have the system OR the EKM into the PKM or have the system replace the PKM with the EKM.

Address space authorization

Each address space owns an authority table. Each table entry defines the PT and SSAR authority that another address space has with respect to the address space that owns the authority table. PT and SSAR authority determine whether an address space can issue PT and SSAR instructions with another address space as the instruction targets. For example, if a service provider's address space has PT and SSAR authority with respect to a user's address space, the service provider can issue PT and SSAR instructions with the target being the user's address space.

Each table entry corresponds to a particular authorization index (AX) value. Therefore, the service provider's AX value corresponds to a specific entry in each user's authority table. That entry defines the service provider's PT and SSAR authority with respect to each user's address space.

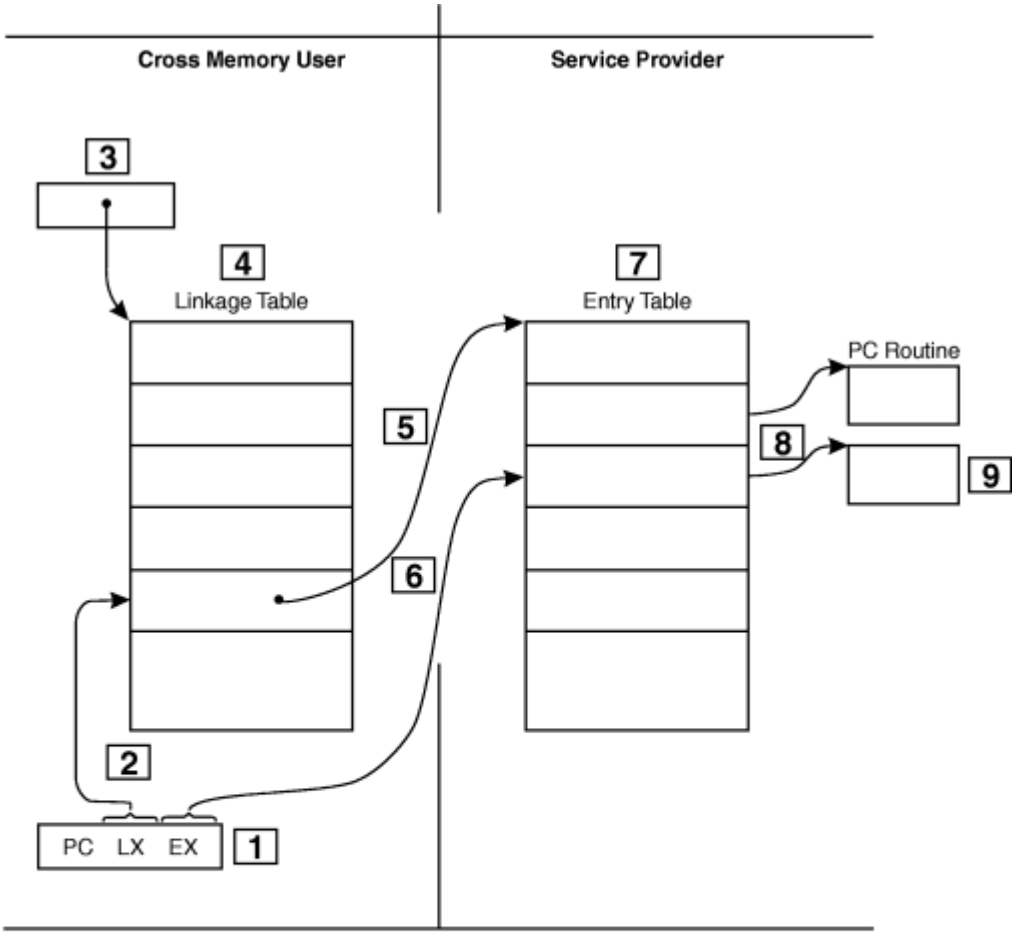
Two AX values, 0 and 1, have the same meaning for all address spaces. A value of 0 always corresponds to an authority table entry that provides neither PT nor SSAR authority. A value of 1 always corresponds to an entry that provides both PT and SSAR authority.

The characteristics of the PC routines defined in the entry table determine whether the service provider needs PT and SSAR authority. The service provider needs the authority if either of the following conditions are true:

- The entry table defines a basic PC routine that causes a space switch
- The entry table defines a stacking PC routine for which the ETDEF macro specifies SASN=OLD.

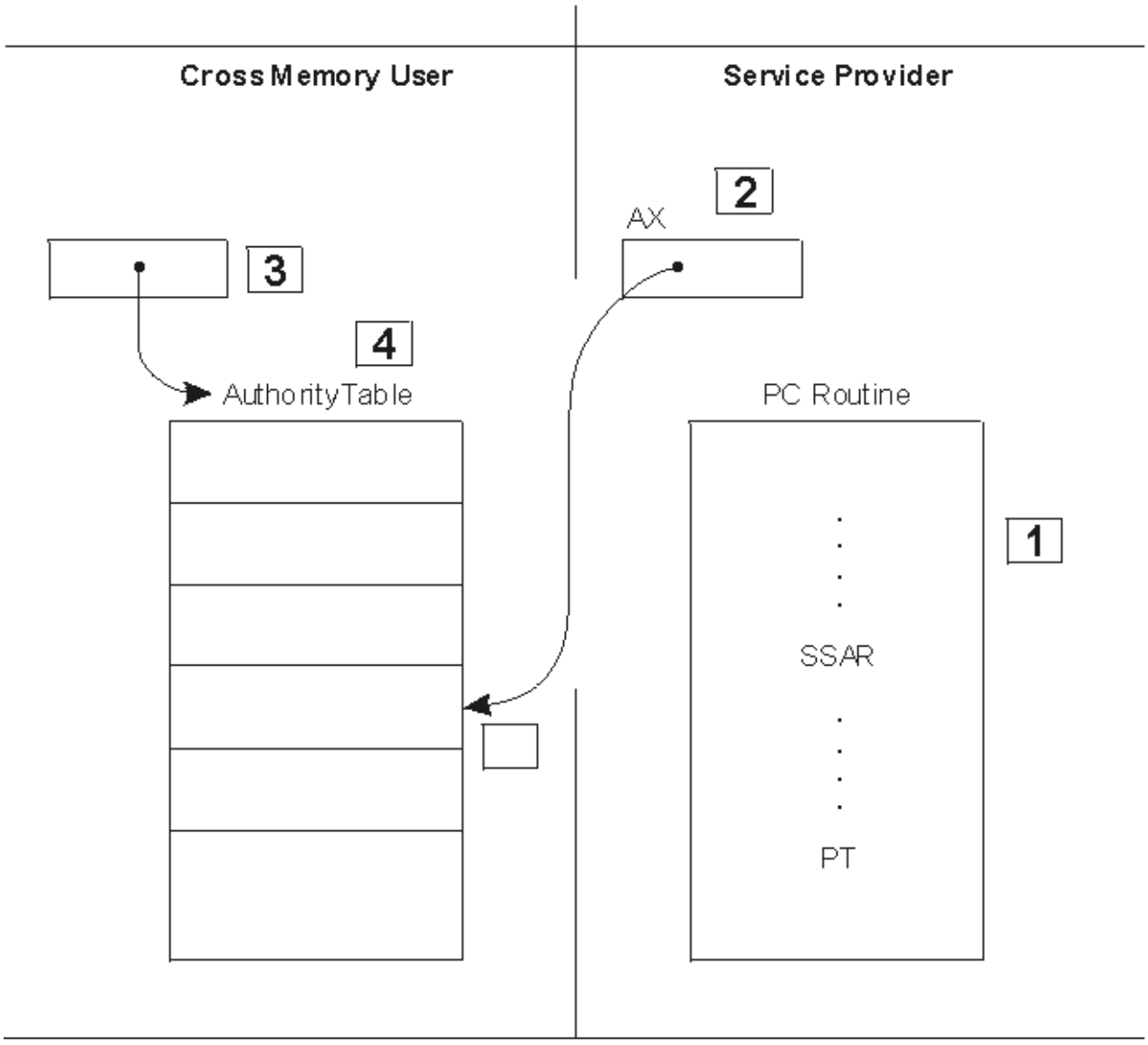
When MVS initially creates an address space, the address space has neither PT nor SSAR authority to any address space. The service provider uses the AXSET and ATSET macros to establish PT and SSAR authority. If a service provider needs PT and SSAR authority to all address spaces, the service provider must issue the AXSET macro and request an AX value of 1.

Figure 9 on page 29 and Figure 10 on page 30 show the environment needed to issue a PC instruction, a PT instruction, or an SSAR instruction. An address space can have only one AX value at any time. The service provider that runs in the address space owns the current AX value for the address space. Only the service provider should set the AX value from 0 to a single nonzero value, or from a nonzero value to 0 in the address space. Other code besides the service provider that runs in the address space should not alter the current AX value, or unpredictable results occur.



- The cross memory user issues the PC instruction 1 .
- The system uses the linkage index 2 together with a system maintained pointer 3 to locate a particular entry in the linkage table 4 .
- The system uses linkage table value 5 together with the EX value 6 to locate a particular entry in the service provider's entry table 7 .
- The system uses information from the entry table entry 8 to invoke the requested PC routine 9 .

Figure 9. PC instruction execution environment



- The service provider issues either a PT or a SSAR 1 instruction.
- The system uses the service provider's AX value 2 together with a system maintained pointer 3 to locate a particular entry in the cross memory user's authority table 4.
- The table entry 5 specifies whether the cross memory user has granted the service provider PT or SSAR authority.

Figure 10. PT and SSAR instruction execution environment

Considerations before using cross memory

Before using cross memory, there are several things of which to be aware about the cross memory environment. The use of cross memory also places some requirements and restrictions on programs that you must consider.

Environmental considerations

- Resource management is different - If your cross memory programs invoke programs in other address spaces, you might need to manage resources differently. For example, your cross memory programs must be able to handle the situation that occurs when an invoked program in another address space abnormally terminates.
- Accounting methods might be affected - Your cross memory programs might acquire the ownership of resources on behalf of cross memory users. You might want to account for these resources differently than the way you account for your own resources.
- The execution time of PC routines is attributed to the home address space, which may not be the address space in which the program executes.

Restrictions

- MVS macros are unavailable to programs running in cross memory mode unless the macro documentation specifically states that it is available.
- Code running in cross memory mode cannot issue an SVC except ABEND. That is, any macro that depends on an SVC is unavailable in cross memory mode.
- Only one step of a job can establish ownership of space switch entry tables. Subsequent job steps cannot issue the LXRES, AXRES, or ETCRE macros.
- Routines that get control as the result of a PC instruction must not use the checkpoint/restart facility.
- In order to be accessed, the address space must be one or more of the following:
  - The home address space
  - A non-swappable address space
  - An address space whose local lock is held or whose local lock is held as a cross memory local (CML) lock.

Requirements

Storage acquired in a cross memory environment is attributed to the job step task of the address space in which it was obtained if the subpool it comes from is task related. A program that acquires such a resource should provide a task termination/address space termination resource manager to clean up any resources obtained on behalf of the terminating task or address space but attributed to another address space's job step task. For more considerations on resource management, see “Resource Management” on page 52.

Establishing cross memory communication

Before cross memory communication can take place, the service provider must establish the cross memory environment. The service provider does this by supplying PC routines and by issuing the MVS macros that establish the necessary linkages and authorizations.

The macros the service provider issues to establish, disconnect, or destroy the cross memory environment are:

- ATSET
- AXEXT
- AXFRE
- AXRES
- AXSET
- ETCON
- ETCRE
- ETDEF

- ETDES
- ETDIS
- LXFRE
- LXRES

The actual set of macros the service provider must issue depends on the following:

- Whether the services will be available to all address spaces or to selected address spaces only
- Whether the PC routine is space switch or non-space switch.

The service provider must issue these macros from a program that is running in supervisor state or with a PSW-key mask of 0-7, and is enabled, unlocked, and in primary ASC mode.

In addition to the previously listed macros, the service provider might also issue the PCLINK macro. The PCLINK macro enables a basic PC routine to save and restore the user's environment. Only basic PC routines that are in supervisor state are permitted to issue the PCLINK macro.

These macros are fully described in one of the following:

- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

Note: Installations that currently build their own PC routine definitions and use IHAETD to map the format 0 ETD may continue to do so. **IBM recommends** the use of the ETDEF macro, however.

Making a PC routine available to all address spaces

If the PC routine is to be available to users in all address spaces, code running in the service provider's address space must issue the macros. Table 4 on page 32 lists the macros that must be issued. For a space switch routine, refer to the first column. For a non-space switch routine, refer to the second column. Several of the macros shown in the table must be issued in a specific sequence. Therefore, **IBM recommends** that the service provider issue the macros in the sequence listed.

Table 4. Macros to issue for a PC routine to be available to all address spaces

Space switch routine	Non-space switch routine
AXSET (See note 1.) LXRES (See note 2.) ETDEF ETCRE ETCON (See note 3.) PCLINK (See note 4.)	LXRES (See note 2.) ETDEF ETCRE ETCON PCLINK (See note 4.)

Notes:

1. Use an AX value of 1.
2. Use a system LX.
3. To determine whether address space authorization (PT and SSAR authority) is needed before issuing the ETCON macro, see “Address space authorization” on page 32.
4. Basic PC routines must issue the PCLINK macro to save and to restore the user's environment.

Address space authorization

The types of PC routines the entry table will define determine whether the service provider needs PT and SSAR authority to the user's address space. PT and SSAR authority are needed if either or both of the following conditions are true:

- The entry table defines a basic PC routine that causes a space switch
- The entry table defines a stacking PC routine for which the ETDEF macro specifies SASN=OLD.

If the service provider needs PT or SSAR authority, it must be obtained before issuing the ETCON macro. The service provider can obtain PT and SSAR authority to all address spaces by issuing AXSET with an AX value of 1. If the correct authorization is not established, the ETCON macro will fail.

Linkage Index

The LXRES macro supplies the service provider with a linkage index. Because the PC routine is to be available to all users, the service provider should obtain a system LX. Note that there are a limited number of system LXs available. You should either be prepared to reuse the system LX if your address space terminates and then restarts, or you should specify REUSABLE=YES on the LXRES macro. See “Reassigning LXs when the LX reuse facility is enabled” on page 56 for more information about system LXs.

PC routines and the entry table

The service provider must also issue the ETDEF macro to define each PC routine and the ETCRE macro to create the entry table. The service provider must then issue ETCON to connect the entry table to the user's address space.

Basic PC routine linkage

When a basic PC routine receives control, it must save the user's environment. Before returning control to the user, the PC routine must restore the user's environment. A basic PC routine that receives control in supervisor state can issue the PCLINK macro to save and to restore the user's environment. A basic PC routine that receives control in problem state must provide code that performs a function similar to the PCLINK macro.

Making a PC routine available to selected address spaces

Characteristics of the PC routines that the entry table defines determines which macros the service provider must issue. Table 5 on page 33 lists the macros that must be issued.

The table is divided into three columns. For a basic PC routine, refer to the first column. For a stacking PC routine where SASN=OLD is specified, refer to the second column. For a stacking PC routine where SASN=NEW is specified, refer to the third column. (U) identifies macros that must be issued by service provider code that is running in the user's address space. Several macros shown in the table must be issued in a specific sequence. Therefore, **IBM recommends** that the service provider issue the macros in the listed sequence.

Table 5. Macros that must be issued for basic and stacking PC routines

Basic PC routine	**Stacking PC routine SASN=OLD**	**Stacking PC routine SASN=NEW**
AXRES AXSET LXRES (See note 1) ETDEF ETCRE ATSET (U) ETCON (U) (See note 2) PCLINK	AXRES AXSET LXRES (See note 1) ETDEF ETCRE ATSET (U) ETCON (U) (See note 2)	LXRES (See note 1) ETDEF ETCRE ETCON (U) (See note 2)

Notes:

1. Use a non-system LX.

2. To determine whether address space authorization (PT and SSAR authority) is needed before issuing the ETCON macro, see “Address space authorization” on page 34 for basic PC routines and “Address space authorization” on page 35 for stacking PC routines.

Linkage index (LX)

Regardless of the types of PC routines that the entry table defines, the service provider must always issue the LXRES macro to reserve a non-system LX. The rest of the macros that the service provider must issue depend on the types of PC routines the entry table will define.

Basic PC Routine

Authorization Index

The service provider must issue the AXRES macro to reserve an authorization index (AX) if address space authorization is required. The service provider must then issue the AXSET macro using the reserved AX value as input. AXSET assigns the AX value as the authorization index for the service provider's home address space.

Address space authorization

The types of PC routines that the entry table defines determines whether the service provider needs PT and SSAR authority to the user's address space. Authority is needed if:

- The entry table defines a basic PC routine that causes a space switch
- The entry table defines a stacking PC routine for which the ETDEF macro specifies SASN=OLD.

If the service provider needs PT or SSAR authority, it must be obtained before issuing the ETCON macro. Otherwise, the ETCON macro will fail.

To obtain address space authorization, service provider code, running in the user's address space, must issue the ATSET macro. Input to ATSET must be the AX value reserved by the service provider.

PC routines and the entry table

The service provider must issue ETDEF to define the PC routines and ETCRE to create the entry table. To connect the entry table to the user's address space, service provider code, running in the user's address space, must issue the ETCON macro.

Basic PC routine linkage

When a basic PC routine receives control, it must issue the PCLINK macro to save the user's environment. Before returning to the user's the PC routine must again issue PCLINK, this time to restore the user's environment.

Stacking PC routine

Authorization index

The types of PC routines that the entry table defines determines whether the service provider must obtain an authorization index. If either of the following conditions are true, the service provider must obtain an authorization index:

- The entry table defines a basic PC routine that causes a space switch
- The entry table defines a stacking PC routine for which the ETDEF macro specifies SASN=OLD.

The service provider must issue the AXRES macro to reserve an AX. The service provider must then issue the AXSET macro using the reserved AX value as input. AXSET assigns the AX value as the authorization index for the service provider's home address space.

Address space authorization

If the service provider had to obtain an authorization index, address space authorization is also required. The service provider must obtain address space authorization (PT and SSAR authority) before issuing the ETCON macro. Otherwise, the ETCON macro will fail.

To obtain address space authorization, service provider code, running in the user's address space, must issue the ATSET macro. Input to ATSET must be the AX value reserved by the service provider.

PC routines and the entry table

The service provider must issue ETDEF to define the PC routines and ETCRE to create the entry table. To connect the entry table to the user's address space, service provider code, running in the user's address space, must issue the ETCON macro.

PC number

Before the user can invoke a basic PC routine or a stacking PC routine, the service provider must supply the user with the PC number that identifies the PC routine. For information about how to do this, see "The PC number" on page 26.

Examples of how to establish a cross memory environment

This topic contains examples that show three ways to establish services that a user can access by issuing a PC instruction. The tasks that the service provider must perform are grouped into the following categories:

- **SETTING UP** initializes the structure that cross memory needs so the transfers of control can take place.
- **ESTABLISHING ACCESS** sets up the linkage the user needs to access the services.
- **PROVIDING SERVICE** consists of designing a service for cross memory use.
- **REMOVING ACCESS** disconnects the linkage that enabled a user to use the services.
- **CLEANING UP** removes the structures established in the setting up step.

"Example 1 - Making services available to selected address spaces" on page 36 shows how a service provider can supply services to users in selected address spaces. (The example shows only one user, but the extra steps for adding users are pointed out.)

"Example 2 - Making services available to all address spaces" on page 43 shows how a service provider can make services available to users in all address spaces.

"Example 3 - Providing non-space switch services" on page 45 explains how a service provider can provide non-space switch services.

For each example, assume that the service provider has obtained common storage that the user can access through name/token callable services. The service provider could use the area pointed to by the token returned by name/token callable services to store the PC numbers corresponding to its services. It could also store some of the lists it needs to invoke PC/AUTH services, and the lists that must be available to different address spaces. Assume also that SERVBLK, shown in Table 6 on page 35, describes the common storage area. All examples use the declared storage areas shown in Table 6 on page 35.

For information on using name/token callable services, see *z/OS MVS Programming: Authorized Assembler Services Guide*.

Table 6. Declared storage for cross memory examples

Declared storage examples			
SERVBLK	DSECT		
LXL	DS	0F	LX LIST

Table 6. Declared storage for cross memory examples (continued)			
Declared storage examples			
LXCOUNT	DS	F	NUMBER OF LXS REQUESTED
LXVALUE	DS	F	LX RETURNED BY LXRES
ELXL	DS	0F	EXTENDED LX LIST
ELXCOUNT	DS	F	NUMBER OF EXTENDED LXS REQUESTED
ELXSEQNO *	DS	F	LX SEQUENCE NUMBER RETURNED BY LXRES
ELXVALUE*	DS	F	LX RETURNED BY LXRES THROUGH THE ELXLIST PARAMETER
AXL	DS	0F	AX LIST
AXCOUNT	DS	H	NUMBER OF AXS REQUESTED
AXVALUE	DS	H	AX RETURNED BY AXES
TKL	DS	0F	TOKEN LIST
TKCOUNT	DS	F	NUMBER OF ETS CREATED
TKVALUE	DS	F	TOKEN RETURNED BY ETCRE
PCTAB	DS	0F	TABLE OF PC NUMBERS
SERV1PC	DS	F	PC NUMBER FOR SERVICE 1
SERV2PC	DS	F	PC NUMBER FOR SERVICE 2

Example 1 - Making services available to selected address spaces

Setting Up

To make its services available to other address spaces through a PC instruction, the service provider sets up the authorization structures and the linkage and entry tables.

To request that the system reserve an authorization index (AX) for the service provider's address space, or an extended authorization index (EAX) for a PC routine, the service provider issues the AXRES macro. The AX or EAX is reserved across the entire system. The home address space at the time the AXRES macro is issued becomes the owner of the AX or EAX:

```
       LA     2,1
       STH    2,AXCOUNT        REQUEST 1 AX
GETAX  AXRES  AXLIST=AXL,RELATED=FREEAX
```

See “Extended authorization index (EAX)” on page 47, “Types of access list entries” on page 102 and “EAX-authority to an address space” on page 115 for more information about the EAX.

To set the AX of the service provider's address space to the AX value that MVS reserved, the service provider issues the AXSET macro:

```
SETAX  AXSET  AX=AXVALUE,RELATED=(GETAX,SETAX)
```

To request that the system reserve a non-system LX for later use, the service provider issues the LXRES macro to reserve a 4-byte LX. A non-system LX allows a service provider to connect to selected users. The home address space at the time the LXRES macro is issued becomes the owner of the LX:

```
         .
         .
         LA      2,1
         ST      2,LXCOUNT          REQUEST 1 LX
GETLX    LXRES   LXLIST=LXL,RELATED=FREELX
```

```
         .
         .
```

To request that the system provide a non-system extended LX value, issue the following:

```
         .
         .
         LA     2,1
         ST     2,ELXCOUNT  REQUEST 1 EXTENDED LX
 GETLX   LXRES  ELXLIST=ELXL,RELATED=FREELX
         .
         .
```

To request that the LX be a reusable extended non-system LX value, issue the following:

```
         .
         .
           LA     2,1
           ST     2,ELXCOUNT  REQUEST 1 EXTENDED LX
GETRLX     LXRES  ELXLIST=ELXL,REUSABLE=YES,RELATED=FREERLX
         .
         .
```

To define which PC routines will be available to user, the service provider must issue two macros, ETDEF and ETCRE. The ETDEF macro builds an entry table descriptor (ETD). Each ETD defines a PC routine. The ETCRE macro uses the ETDs as input to build an entry table. The entry table contains ETD entries for each of the PC routines that the service provider is making available to the user. The home address space at the time the service provider issues the ETCRE macro becomes the owner of the entry table.

There are two ways the service provider can use the ETDEF macro:

- If all of the information about the PC routine being defined is available at the time the ETDEF macro is assembled, the service provider can statically define an ETD by specifying the TYPE=ENTRY option.
- If some of the information about the PC routine being defined is unavailable when assembling the ETDEF macro, the service provider must issue ETDEF twice: once with TYPE=ENTRY, and once with TYPE=SET. TYPE=ENTRY reserves storage for an ETD entry. TYPE=SET initializes the ETD entry, and overrides any options specified on TYPE=ENTRY. For any options the service provider omits on TYPE=SET, the system uses the default values.

  Example: if the service provider specifies TYPE=ENTRY with ASYNCH=NO, and then does not specify the ASYNCH parameter on TYPE=SET, the system uses the default of ASYNCH=YES. (See the description of the ETDEF macro in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for more information.)

  Use this method if any of the input data is unresolved when assembling the ETDEF macro. For example, a program's *name* may be known at assembly time, but not the address at which it will be loaded.

To define a complete ETD suitable as input to ETCRE, the service provider must issue the ETDEF macro three or more times:

- Once to define the beginning of the table
- Once for each PC routine to be defined in the table
- Once to define the end of the table.

Note: Instead of issuing ETDEF, the service provider has the option to code the data areas that ETDEF builds. IBM provides a mapping macro, IHAETD, that maps the format 0 ETD. **IBM recommends**, however, the use of the ETDEF macro.

The following figure shows how to use ETCRE and ETDEF to create an entry table that defines two stacking PC routines. This example works only when the PC routines are located in LPA or in the nucleus.

```
********* Executable Instructions
*
         .
         .
CET1     ETCRE ENTRIES=ETDESC,RELATED=(CONET,DISET1,DESET1)
         ST    0,TKVALUE      Save Returned Token
         .
         .
********* Data Constants
*
ETDESC   ETDEF TYPE=INITIAL,RELATED=(CET1)
         ETDEF TYPE=ENTRY,PROGRAM='SERVICE1',SSWITCH=YES,             X
               STATE=PROBLEM,AKM=(0:15),EKM=8,EK=8,PKM=REPLACE
         ETDEF TYPE=ENTRY,PROGRAM='SERVICE2',SSWITCH=NO,              X
               STATE=SUPERVISOR,AKM=(0:15),EKM=(0:15),PKM=OR
         ETDEF TYPE=FINAL
```

Figure 11. Using ETDEF to statically define entry table descriptors

In the previous example, the first ETDEF macro defines the beginning of the entry table definition.

The second ETDEF macro defines a space switch PC routine named *Service1*. This PC routine receives control in problem state, requires that all input and output parameters be in key 8 storage, and can reference date that is in key 8 storage only. This example of the ETDEF macro shows how to define a stacking PC routine that *decreases* authority.

- The routine is a stacking PC because PC=STACKING is the default.
- The STATE=PROBLEM parameter specifies that the PC routine will receive control in problem state.
- The parameter AKM=(0:15) specifies that programs running with any PSW key may invoke the PC routine.
- The parameter EK=8 specifies that the PC routine will run with PSW key 8.
- The parameter PKM=REPLACE specifies that the system is to replace the PSW-key mask with the mask specified by the parameter EKM=8 before invoking the PC routine.

The third ETDEF macro defines a non-space switch PC routine named *Service2*. This PC routine can reference input/output parameters in any key. This example of the ETDEF macro shows how to define a stacking PC routine that *increases* authority.

- The routine is a stacking PC because PC=STACKING is the default.
- The STATE=SUPERVISOR parameter specifies that the PC routine, *Service2*, will receive control in supervisor state.
- The parameter AKM=(0:15) specifies that programs running with any PSW key may invoke the PC routine.
- The parameter EKM=(0:15) specifies that the program will run with all PKM bits on.
- The parameter PKM=OR specifies that the system is to OR the PSW key mask of the caller with the mask specified by EKM=(0:15) before invoking the PC routine.

The last ETDEF macro defines the end of this entry table definition.

When a PC routine is not in LPA and is not in the nucleus, the service provider will not know the location of the PC routine until it is loaded. Also, the service provider will not know the address of the PC routine's associated recovery routine (ARR) until it is loaded, and will not know the EAX value until the AXRES macro is issued. Therefore, the service provider must create at least part of the entry table definitions dynamically. The following figure shows how the service provider could create the entry table if the PC routine, *Service1*, and the ARR, *ARR1*, were loaded into private storage first. The figure shows code for a non-reentrant program. ETDEF TYPE=SET specifies a *complete* entry replacement. All options are either set or defaulted. Nothing is carried over from the TYPE=ENTRY declaration. Note that, in this example, the service provider uses the AX value, provided through the AXRES macro, as an EAX value.

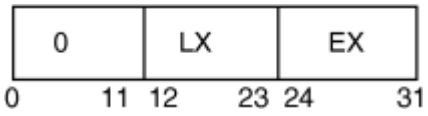
```
*
********* Executable Instructions
*
         LA   1,1
         STH  1,AXNUM
         AXRES AXLIST=AXL              GET AN EAX FOR SERVICE1 (THE     X
                                             PC ROUTINE)
         LH   4,AXVAL
         LOAD EP=SERVICE1              GET ADDRESS OF SERVICE1
         ST   0,SRV1ADDR               SAVE ADDRESS OF SERVICE1
         LR   2,0
         LOAD EP=ARR1                  GET ADDRESS OF ARR1
         ST   0,ARR1ADDR               SAVE ADDRESS OF ARR1
         LR   3,0
         .
         .
         ETDEF TYPE=SET,ETEADR=ETD1,ROUTINE=(2),SSWITCH=YES,          X
               STATE=PROBLEM,AKM=(0:15),EKM=8,EK=8,PKM=REPLACE,       X
               ARR=(3),EAX=(4)
         .
         .
CET1     ETCRE ENTRIES=ETDESC,RELATED=(CONET,DISET1,DESET1)
         ST    0,TKVALUE               SAVE RETURNED TOKEN
*
********* Data Definition area
*
SRV1ADDR DS F                          ADDRESS OF SERVICE1
ARR1ADDR DS F                          ADDRESS OF ARR1
ETDESC   ETDEF TYPE=INITIAL
ETD1     ETDEF TYPE=ENTRY,ROUTINE=0
ETD2     ETDEF TYPE=ENTRY,PROGRAM='SERVICE2',SSWITCH=NO,              X
               STATE=SUPERVISOR,AKM=(0:15),EKM=(0:15),PKM=OR
         ETDEF TYPE=FINAL
AXL      DS 0F                         AXLIST
AXNUM    DS H                          NUMBER OF AXs REQUESTED
AXVAL    DS H                          RETURNED AX (OR EAX)
```

Figure 12. Using ETDEF to dynamically define entry table descriptors

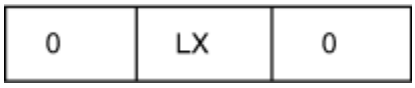
The preceding example of the ETDEF macro shows how to define a stacking PC routine that uses an ARR and an EAX:

- The parameter ROUTINE=(2) specifies that the PC entry point address is in register 2.
- The parameter ARR=(3) specifies that the address of the ARR to receive control if the stacking PC routine ends abnormally is in register 3.
- The parameter EAX=(4) specifies that the EAX value for the PC routine is in register 4.

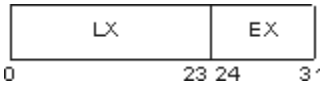
Once the linkage and entry tables have been created, the service provider can construct the PC numbers that identify the PC routines. A PC number is a fullword value formed by combining an LX and an EX.



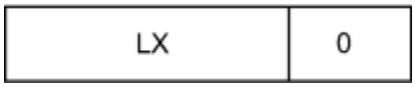
The LXRES macro returns the LX in the format that's shown below. This format allows the service provider to OR the LX with an EX to form a PC number:



Shown below is an LX that is available only when the LX reuse facility is enabled:



The LXRES macro returns the LX in the format that's shown below. This format allows the service provider to OR the LX with an EX to form a PC number:




```
          L   2,LXVALUE        LX=PC# WITH EX OF 0
          ST  2,SERV1PC        SAVE EX=0 PC# FOR FIRST SERVICE
          LA  2,1(,2)          CONSTRUCT EX=1 PC#
          ST  2,SERV2PC        SAVE PC# FOR SECOND SERVICE
```

To make the PC numbers accessible, the service provider can save the address of its SERVBLK by using name/token callable services:

```
          LA   2,SERVBLK
          ST   2,SERVBLKA
          CALL IEANTCR,(LEVEL,NAME,TOKEN,PERSOPT,RETCODE)
          .
          .
          .
LEVEL     DC   A(IEANT_SYSTEM_LEVEL)
NAME      DC   CL16'SERVBLK'
TOKEN     DS   0XL16
SERVBLKA  DS   A
          DS   XL12
PERSOPT   DC   A(IEANT_NOPERSIST)
RETCODE   DS   F
          IEANTASM             INCLUDE NAME/TOKEN SERVICES          X
                               ASSEMBLER DECLARATION STATEMENTS
```

Establishing access

The next two steps make the service provider's services available to a user. The instructions used for these two steps must be issued from the user's address space by a program running in supervisor state or with a PKM value of 0-7. If the user is a problem state program, the service provider must provide code that executes on behalf of the user with the user's address space as the home address space. The service provider must repeat these two steps for each user.

1. Set the PT and SSAR authority in the user's authority table entry that corresponds to the service provider's AX value. This action allows the service provider to issue a PT or SSAR instruction with the user's address space as the instruction target.

   ```
   SETAT  ATSET  AX=AXVALUE,PT=YES,SSAR=YES,RELATED=RESETAT
   ```

2. Connect the service provider's entry table to the user's linkage table at the entry that corresponds to the service provider's LX. After the system completes the connection, the linkage table entry points to the service provider's entry table.

   ```
            LA      2,1
            ST      2,TKCOUNT    SET COUNT OF ETS TO BE CONNECTED
   CONET    ETCON   TKLIST=TKL,LXLIST=LXL,RELATED=DISET1
   ```

   Or connect the service provider's entry table to the linkage tables at the entry that corresponds to the service provider's extended LX values. After the system completes the connection, the linkage table entry points to the service provider's entry table.

   ```
            LA      2,1
            ST      2,TKCOUNT    SET COUNT OF ETS TO BE CONNECTED
   CONETX   ETCON   TKLIST=TKL,ELXLIST=ELXL,RELATED=DISETX
   ```

To invoke a PC routine, the user still needs a PC number. The service provider and the user must have previously agreed on a method the service provider will use to provide a PC number. For example, the service provider could provide a macro that the user issues to find the PC number that the service provider has stored in a table in commonly addressable storage. In addition, if a reusable LX was reserved by the service provider, the LX sequence number associated with the LX/PC number must also be provided to the user. The service provider could use the same macro that the user issued to find the PC number, to provide the LX sequence number.

At this point in the example, the service provider has provided two services that the user can access using PC instructions. The service provider has also established authority to issue PT and SSAR instructions to the user's address space. The user's linkage table is connected to the service provider's entry table as shown in Figure 13 on page 41.

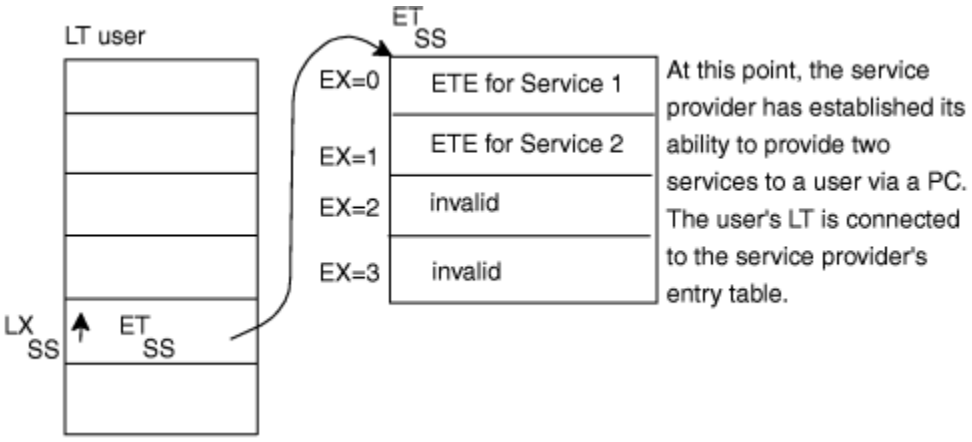


Figure 13. Linkage table and entry table connection

Invoking a PC routine

The PC instruction gives control to a PC routine. The PC number determines the specific PC routine that receives control. The entry table entry that corresponds to the PC number defines the PC routine's location and environment. To return to the caller, a stacking PC routine issues the PR instruction; a basic PC routine issues the PT instruction.

Figure 14 on page 41 shows the instruction sequence needed to invoke a stacking PC routine. The stacking PC automatically saves the user's environment. When the PC routine issues the PR instruction to return control to the caller, the system restores the caller's environment.

Note: Getting the LX sequence number and putting it in the high-order half of register 15 is necessary only for a reusable LXs, but it will not interfere with a non-reusable LX. For more information on reusable LXs, see "Reassigning LXs when the LX reuse facility is enabled" on page 56.

```
 .
 .
 USING PSA,0
 CALL  IEANTRT,(LEVEL,NAME,TOKEN,RETCODE)  OBTAIN SERVBLK ADDRESS
 CLC   RETCODE,=A(IEANT_OK)     CHECK RETURN CODE
 BNE   NOSERVIC                 IF NO TOKEN, SERVICES NOT AVAILABLE
 L     14,SERVBLKA
 USING SERVBLK,14               ACCESS SERVBLK
 LMH   15,15,SERV1LXSEQNO       GET LX SEQUENCE NUMBER AND PUT IN HIGH-ORDER

*
          HALF (BITS 0-31) OF REG 15
 L     14,SERV1PC               GET PC NUMBER
 DROP  14
 PC    0(14)                    ISSUE THE PC
 .
 .
 LEVEL    DC A(IEANT_SYSTEM_LEVEL)
 NAME     DC CL16'SERVBLK'
 TOKEN    DS 0CL16
 SERVBLK  DS 0F
 SERV1PC  DS F
 SERV1LXSEQNO  DS F RETCODE  DS F
          IEANTASM              INCLUDE NAME/TOKEN SERVICES         X
                                    ASSEMBLER DECLARATION STATEMENTS
```

Figure 14. Calling sequence for a stacking PC routine

Figure 15 on page 42 shows the instruction sequence needed to invoke a basic PC routine. The calling program must save registers and its SASID before issuing the PC instruction. When the PC returns control, the caller must restore registers and the SASID.

```
.
.
STM   14,12,12(13)            SAVE REGISTERS
ESAR  2                       SAVE CALLER'S SASID IN THE
ST    2,16(,13)               REG 15 SLOT OF SAVEAREA
USING PSA,0
CALL  IEANTRT,(LEVEL,NAME,TOKEN,RETCODE)  OBTAIN SERVBLK ADDRESS
CLC   RETCODE,=A(IEANT_OK)    CHECK RETURN CODE
BNE   NOSERVIC                IF NO TOKEN, SERVICES NOT AVAILABLE
L     15,SERVBLKA
USING SERVBLK,15              ACCESS SERVBLK
L     2,SERV1PC               OBTAIN SERVICE1 PC NUMBER
DROP  15
PC    0(2)                    ISSUE THE PC
L     14,12(,13)              RESTORE REG 14
L     2,16(,13)               LOAD SAVED SASID
SSAR  2                       RESTORE CALLER'S SASID
LM    2,12,28(13)             RESTORE REGS 2-12
.
.
LEVEL    DC A(IEANT_SYSTEM_LEVEL)
NAME     DC CL16'SERVBLK'
TOKEN    DS 0CL16
SERVBLKA DS A
         DS XL12
RETCODE  DS F
         IEANTASM             INCLUDE NAME/TOKEN SERVICES        X
                                  ASSEMBLER DECLARATION STATEMENTS
```

Figure 15. Calling sequence for a basic PC routine

To make it easier for the user to invoke a PC routine, the service provider can provide a macro that generates the needed instruction sequence.

Removing Access

The next two steps remove access to previously provided services. The steps must be performed with the user's address space as the home address space. These steps are essentially the opposite of the steps used to establish access. First, the service provider removes PT and SSAR authority to the user's address space.

```
RESETAT    ATSET AX=AXVALUE,PT=NO,SSAR=NO,RELATED=(SETAT)
```

The service provider then disconnects the entry table from the user's linkage table.

```
DISET1     ETDIS  TKLIST=TKL,RELATED=CONET
```

Cleaning Up

Before shutting down, the service provider must remove all cross memory connections and release any cross memory resources it owns. After ensuring that all connections to the entry table have been disconnected, the service provider destroys the entry table.

```
DESET1     ETDES  TOKEN=TKVALUE,RELATED=CET1
```

The service provider then frees the LX so it will be available for reuse. If this is not a reusable LX, the address space will only be available for reuse if no other address space is connected to the LX.

Non-extended LXLIST

```
FREELX     LXFRE  LXLIST=LXL,RELATED=LXRES
```

Extended LXLIST

```
FREELX     LXFRE  ELXLIST=ELXL,RELATED=ELXRES
```

The service provider then resets the AX of its address space to zero.

```
           SR     2,2                   ZERO VALUE
RESETAX    AXSET  AX=(2),RELATED=LXRES  RESET AX TO ZERO
```

Finally, the service provider frees the AX so the system can reuse it. Freeing the AX removes PT and SSAR authority corresponding to the service provider's AX in all authority tables in the system.

```
FREEAX     AXFRE   AXLIST=AXL,RELATED=GETAX
```

Example 2 - Making services available to all address spaces

This example shows how a service provider makes services available to all users. The example uses the same storage areas as Example 1 (see Table 6 on page 35 for the storage areas used), however, it does not need the AX list. The main difference between Example 1 and Example 2 is Example 2's use of a system LX and an AX value of 1. A system LX allows the service provider to connect an entry table to all address spaces, and the AX gives the service provider PT and SSAR authority to all address spaces.

Note: Do not use a system LX unless your service is intended for all address spaces. Establishing a system LX makes the ASID of the address space unusable until the next IPL. For more information about ASID reuse, see “Reusing ASIDs” on page 53.

Setting up

The service provider first obtains a system LX. MVS sets aside part of the available LXs for use as system LXs. When the service provider connects an entry table to a system LX, the entry table is connected to all present and future address spaces.

Unlike a non-system LX, a non-reusable system LX cannot be freed for reuse. When an address space that owns a non-reusable system LX terminates, the LX becomes dormant. The system allows a dormant system LX to be reconnected to an address space different from the original owning address space. This is an important consideration for a service provider that can be terminated and then restarted. The service provider must have a way to remember the non-reusable system LX it owned so that it can connect the LX to an entry table when it is restarted. A reusable system LX, on the other hand, can be freed for reuse. See “Reassigning LXs when the LX reuse facility is enabled” on page 56 for more information.

In the example, the service provider would first test LXVALUE. If LXVALUE was zero, the service provider would issue the LXRES macro. Otherwise the service provider would pass the value found in LXVALUE to the ETCON macro.

The code shown in the following three steps runs with the service provider's address space as the home address space. The first step obtains a system LX. If the service provider's address space is coming up for the first time since IPL, the service provider issues the LXRES macro with the SYSTEM=YES option. The service provider must then save the LX somewhere, probably in common storage, so it is accessible if the service provider is restarted.

```
           LA     2,1
           ST     2,LXCOUNT     REQUEST 1 SYSTEM LX
GETSLX     LXRES  LXLIST=LXL,SYSTEM=YES
```

The service provider then sets its address space AX to a value of 1. An AX value of 1 authorizes the service provider to issue a PT or SSAR instruction to all other address spaces. (Because the service provider is providing a service to all users, the service provider does not need to obtain a unique AX.)

```
           LA     2,1
           AXSET  AX=(2)
```

The service provider then issues the ETCRE macro to create the entry table.

```
         ETCRE  ENTRIES=ETDESC
         ST     0,TKVALUE          SAVE THE ET TOKEN
```

The service provider can construct the PC numbers and make them accessible the same way it did in “Example 1 - Making services available to selected address spaces” on page 36.

Establishing access

To connect the entry table to the linkage table in each current and future address space, the service provider issues the ETCON macro. In this case, the service provider can issue the ETCON macro from any address space.

```
         LA     2,1
         ST     2,TKCOUNT      SET COUNT OF ETS TO BE CONNECTED
         ETCON  LXLIST=LXL,TKLIST=TKL
```

All address spaces in the system now have access to the service provider's services. Figure 16 on page 44shows how the linkage and entry tables appear at this point.

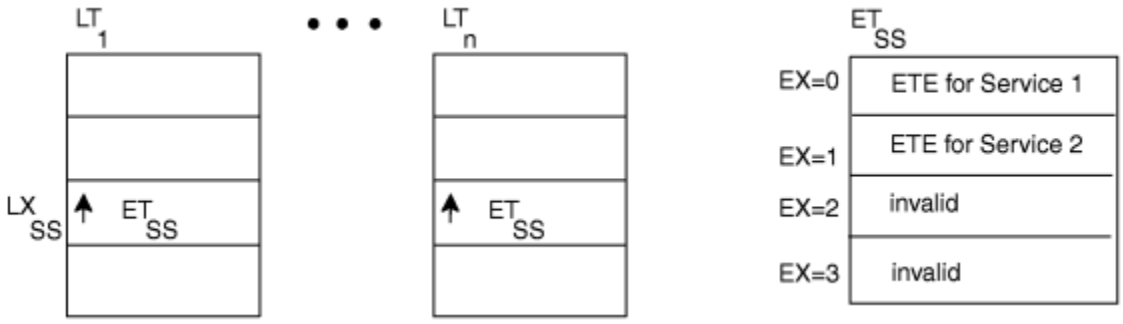


Figure 16. Linkage and entry tables for a global service

Providing Service

The service provider supplies services in the same way as in Example 1. The users of the services must be aware of the PC number associated with each service.

Removing Access

To remove access, the service provider disconnects all users and destroys the entry table by issuing the ETDES macro with the PURGE=YES option. This disconnects the entry table from all linkage tables in the system and then destroys it. For information about how the system reuses the system LX, see “Reassigning LXs when the LX reuse facility is enabled” on page 56. (ETDIS cannot be used to disconnect an entry table that is connected to a system LX.)

```
         ETDES  TOKEN=TKVALUE,PURGE=YES
```

Cleaning Up

Finally, the service provider sets the AX of its address space to 0.

```
         SR     2,2
         AXSET  AX=(2)
```

Example 3 - Providing non-space switch services

This example is similar to “Example 1 - Making services available to selected address spaces” on page 36 except the service provider will provide only non-space switch PC routines. The service provider code will be the same as in example 1 or 2 with the following exceptions:

- The ETDEF macros that define PC routines will all specify SSWITCH=NO.
- The service provider will not issue the AXRES, AXSET, or ATSET macros.

PC linkages and PC routine characteristics

When a user issues a PC instruction, the system transfers control to a PC routine. An entry in the service provider's entry table defines the PC routine that is to receive control. Data in this entry also determines the type of linkage the system will use to invoke the PC routine. The types of linkages are *stacking* and *basic*.

The stacking and basic PC linkages share some common capabilities. The stacking linkage, however, offers more capability and provides better performance than does the basic linkage. The service provider uses the PC parameter on the ETDEF macro to define the type of linkage that will be used. Stacking is the default. **IBM recommends** the use of the stacking PC linkage. This document refers to a PC routine as either a *stacking PC routine* or a *basic PC routine* depending on the linkage used to invoke the routine.

PC linkage capabilities

The stacking PC linkage and the basic PC linkage provide the following capabilities:

- The PC routine's PKM authority can be increased.
- Basic PC routines must receive control in primary mode; stacking PC routines have the option to do so.
- Basic PC routines must receive control with SASN=old PASN; stacking PC routines have the option to do so.
- The PC routine can receive control in either problem state or supervisor state.
- The PC routine can be either a space switch routine or a non-space switch routine.

The stacking PC linkage also provides the following additional capabilities that the basic PC linkage does not provide:

- The PC routine's PKM authority can be decreased.
- The PC routine's PSW key can be set from data in the entry table.
- The PC routine can receive control in AR mode.
- An entry point to an associated recovery routine (ARR) can be defined in the entry table.
- The system automatically uses the linkage stack to save and restore the user's environment.

Defining a PC routine

When you define a PC routine, you define its operating characteristics and its environment. Several definitions apply to both basic and stacking PC routines. Other definitions apply to stacking PC routines only.

For each PC routine, you must specify the type of linkage, basic or stacking, that the system is to use when a user invokes the routine. **IBM recommends** that you use only the stacking linkage. To define the type of linkage, use the PC keyword on the ETDEF macro.

Note: If you currently provide basic PC routines, you may continue to use these basic PC routines without change.

All of the information that you provide to define a basic PC routine you also provide to define a stacking PC routine. There is also additional information that you can provide for stacking PC routines only. The topic

"Definitions common to both stacking and basic PC routines" on page 46 explains how to provide the definitions common to both types of PC routines. The topic "Definitions for stacking PC routines only" on page 47 explains how to provide the definitions that apply to stacking PC routines only.

Definitions common to both stacking and basic PC routines

The PC routine definitions explained in this topic apply to both stacking and basic PC routines. For each PC routine you must define:

- Whether the PC routine will receive control in supervisor state or problem state
- Whether the PC routine is a space switch routine or a non-space switch routine
- The PSW key-mask (PKM) that a problem state program must have in order to invoke the PC routine
- The addressing mode of the PC routine if you specified the ROUTINE parameter on the ETDEF macro
- Whether the PC routine will run under the user's PSW-key mask or under a different PSW-key mask

Supervisor state or problem state

A PC routine can receive control in either supervisor state or problem state. A PC routine must receive control in supervisor state if:

- The PC routine uses system services that are available only to programs that run in supervisor state. An example of such a service is the PCLINK macro.
- The PC routine can be invoked by a basic PC issued from a program that runs in supervisor state. This requirement exists because the system does not allow a basic PC routine that receives control in problem state to issue the PT instruction to return to a program that runs in supervisor state.

Otherwise, the PC routine can run in problem state.

To specify whether a PC routine receives control in supervisor state or in problem state, use the STATE parameter on the ETDEF macro. The default is to receive control in problem state.

Space switch or non-space switch

You can define a PC routine as either a space switch routine or a non-space switch routine. When making this decision, consider the nature of the PC routine and the data it manipulates.

Use a non-space switch PC routine if the PC routine must support problem state callers and must run in supervisor state in the caller's address space. If you do not have these requirements, you can use a space switch routine, which has certain advantages. A space switch routine:

- Provides code isolation
- Allows you to access data in multiple address spaces
- Prevents you from having to place your code in common storage.

To define a PC routine as either a space switch or non-space switch routine, specify the SSWITCH parameter on the ETDEF macro. The default is to define the routine as a non-space switch routine.

Problem state program authorization

You can specify the PSW key mask (PKM) that a problem state program must be running under in order to invoke a PC routine. When a program in problem state issues a PC instruction, the system uses the program's PKM and the PC routine's authorization key mask (AKM) to determine whether the program is authorized to invoke the PC routine. If any bit in the program's PKM is on and the corresponding bit in the AKM is also on, the program is authorized and the system invokes the PC routine. Otherwise, the system disallows the invocation.

To define the AKM, specify the AKM parameter on the ETDEF macro.

Addressing mode

A PC routine can receive control in either 24-bit addressing mode or 31-bit addressing mode. If you specify the ROUTINE parameter on the ETDEF macro, you can specify RAMODE on ETDEF to indicate the PC routine's addressing mode. If you specify the PROGRAM parameter on ETDEF, then the system locates the PC routine and determines its addressing mode. The default is to pass control to the PC routine in 31-bit addressing mode.

PSW-key mask (PKM)

You can specify the PKM that a PC routine is to run under. The PKM, which has meaning only for PC routines that run in problem state, defines:

- The PSW key values that the PC routine can set by means of the MODESET macro or the SPKA instruction
- Whether the PC routine is authorized to use the MVCK, MVCS, and MVCP instructions
- Other PC routines that the PC routine can invoke

Basic PC routines and stacking PC routines can run under the user's PKM, or they can run under a PKM that provides greater authority than does the user's PKM. The EKM and PKM parameters on the ETDEF macro enable you to define the PKM the PC routine will run under.

If the user's PKM provides sufficient authority for the PC routine, use the user's PKM by omitting the EKM parameter from the ETDEF macro.

If the PC routine needs more authority than the user has, use the EKM parameter to increase the authority. You must also omit the PKM parameter or specify PKM=OR. When you specify PKM=OR or omit PKM, the system determines the PKM authority for the PC routine by ORing the caller's PKM value with the EKM value.

For a stacking PC routine only, you can decrease authority or define a new authority. You do this by defining the authority in the EKM value and specifying PKM=REPLACE. Specifying PKM=REPLACE causes the system to use the EKM value as the new PKM value for the PC routine.

Definitions for stacking PC routines only

In addition to the previously discussed definitions, for each stacking routine you can define:

- The ASC mode of the PC routine
- The PC routine's extended authorization index (EAX)
- The value that SASN is to assume after the PC instruction executes
- The address of an associated recovery routine (ARR)
- The PSW key under which the PC routine is to execute

ASC mode

A stacking PC routine can receive control in either primary address space control (ASC) mode or in AR ASC mode. The ASC mode determines whether the PC routine can use ARs. AR ASC mode is required to use ARs. The ASCMODE parameter on the ETDEF macro determines the mode. The default is for the PC routine to receive control in primary ASC mode.

Extended authorization index (EAX)

A stacking PC routine can receive control with the same extended authorization index (EAX) value as the user who issued the PC instruction, or with a new EAX value. The EAX controls authorization for the PC routine to use access list entries. To specify a new EAX, use the EAX parameter on the ETDEF macro. The default is for the PC routine to use the user's EAX. For more information about the function of the EAX, see “Types of access list entries” on page 102.

SASN value

A stacking PC routine can receive control with the secondary address space number (SASN) set to one of two values:

- SASN can equal the number of the user's primary address space (the address space from which the PC instruction was issued).
- SASN can equal the number of the service provider's address space (the address space where the PC routine executes)

The SASN parameter on the ETDEF macro determines the SASN value. The default is for SASN to equal the number of the user's primary address space.

Here are two examples of how you might use the SASN parameter:

- If the PC routine does not run in AR mode and you want to access data in the user's address space (by using the MVCP or MVCS instructions), specify SASN=OLD. This will give the PC routine the authority it needs to issue those two instructions.
- If you do not need or want the PC routine to have secondary authority to the user's address space, specify SASN=NEW.

Associated recovery routine (ARR)

A stacking PC routine can identify an ARR that is to receive control if the PC routine encounters an error. An ARR enables a stacking PC routine to avoid the overhead of defining and activating a recovery routine each time it's invoked. See the section on providing recovery in *z/OS MVS Programming: Authorized Assembler Services Guide* for more information about ARRs.

PSW key

By default, a PC routine runs under the caller's PSW key. You have the option to run under a different key. To specify a different PSW key, use the EK parameter on the ETDEF macro.

PC routine requirements

All PC routines must meet certain requirements depending on the type of PC routine, stacking or basic.

Stacking PC routines

Stacking PC routines must meet the following requirements:

- They must not use the checkpoint/restart facility.
- They must be either permanently resident in LPA or the nucleus, or they must be loaded under the job step task of the address space that created the entry table.
- They must issue the PR instruction to return control to the user.
- Stacking PC routines that cause a space switch must:
  - Run in an address space that is non-swappable
  - Use only those MVS services that are supported in cross memory mode.

Basic PC routines

Basic PC routines must meet the following requirements:

- They must not use the checkpoint/restart facility.
- They must be either permanently resident in LPA or the nucleus, or they must be loaded under the job step task of the address space that created the entry table.
- They must use the PCLINK macro or provide code to save and restore the user's environment.
- They must use the PT instruction to return control to the user.

- Basic PC routines that cause a space switch must:
  - Run in an address space that is non-swappable
  - Use only those MVS services that are supported in cross memory mode.

Linkage conventions

There are linkage conventions that user programs must observe and linkage conventions that PC routines must observe. These conventions vary depending on the type of PC linkage used, basic or stacking, and the ASC mode of the programs. For basic PC routines, receiving control in supervisor state requires the use of different conventions than does receiving control in problem state.

Basic PC

A basic PC routine receives control in primary mode and only from a user program that's running in primary mode. The PC routine can receive control in either problem state or in supervisor state. In order to return control to the user's program, the PC routine must save the user's environment. Before issuing the PT instruction to return to the user, the PC routine must restore the previously saved environment.

User program

Before issuing a PC instruction, the user's program must:

- Save general registers 14 through 12 at the location starting at offset 12 (word 4) in the save area pointed to by general register 13. The program must save registers *before* issuing a PC instruction because the basic PC linkage updates general registers 3, 4, and 14. As a result of the update, the address space where the save area resides might no longer be the currently addressable address space.
- Save the current SASID in bits 16-31 of save area word 5.
- Optionally load general registers 0, 1, and 15 as parameter registers.
- Load general register 2 with the PC number.
- Issue the PC instruction specifying general register 2.

When the PC routine returns control to the user's program, the user's program must restore its general registers and its secondary address space identifier (SASID).

PC routine that receives control in supervisor state

A basic PC routine that receives control in supervisor state can use the PCLINK macro to save and restore the user's environment. Although the use of PCLINK is optional, **IBM recommends** its use. A PC routine that does not use the PCLINK macro must provide a method of saving and restoring the users environment.

The PCLINK STACK macro saves the following information:

- Caller's save area address from caller's general register 13
- AMODE, return address, and PSW problem state bit from caller's general register 14
- Parameter registers: general registers 0, 1, and 15
- Caller's PSW key and other information from caller's general register 2 as follows:

  In bits 0-23, bits 8-31 of caller's general register 2
  In bits 24-27, PSW key
  In bits 28-31, zeroes
- Caller's PSW key mask and PASID from caller's general register 3
- Latent parameter list address for this entry from caller's general register 4
- Return address from the PCLINK service routine to the program that issued PCLINK STACK. This point is just after the PC routine entry point.
- Program mask from current PSW.

After issuing PCLINK STACK, the PC routine can begin processing. If necessary, the PC routine can use the PCLINK macro with the EXTRACT option to get information from the PCLINK stack.

When the PC routine is ready to return control to the user's program, the PC routine must load into general registers 0, 1, and 15 any data to be returned to the user. The PC routine can then issue PCLINK with the UNSTACK,THRU option. This option restores general registers 3, 13, 14, the program mask and, optionally, the original PSW protection key. The PC routine can then issue the PT instruction to return control.

For information about coding the PCLINK macro, see *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

PC Routine That Receives Control In Problem State

A basic PC routine that receives control in problem state must provide a method for saving and restoring the user's environment. The PC routine cannot use the PCLINK macro because that macro works only in supervisor state. **IBM recommends** that a PC routine that receives control in problem state use the stacking PC linkage.

Stacking PC

A stacking PC routine can receive control in either primary mode or AR mode. The user program that issues the PC instruction can be in either primary mode or AR mode. When the user's program issues the PC instruction, the system saves the user's environment on the linkage stack. When the PC routine issues the PR instruction to return to the user, the system restores the user's environment from the stack before returning control. Thus, there is no need for the caller or the PC routine to either save or restore the environment. The system saves the caller's general registers (0 - 15), ARs (0 - 15), PASN, SASN, PKM, and PSW. If necessary, the PC routine can issue the extract stacked state instruction (ESTA) to examine the stacked entry.

User in Primary Mode

A user program that's running in primary mode can invoke a PC routine that receives control in either primary mode or AR mode. Any addresses that the user program passes must be located within the user's primary address space. The user must not use ARs to pass parameter values or addresses. Before issuing a PC instruction, the user must:

- Load the PC number into general register 14.
- If there is a parameter list to pass, load its address into general register 1.

When the PC routine returns control, GPRs 2 - 13 and ARs 2 - 13 are restored to their original values. GPRs 0, 1, and 15, and ARs 0, 1, and 15 contain the values that were in them when the PC routine issued the PR instruction. GPR 14 and AR 14 are used as work registers by the system.

User In AR Mode

A user program that's running in AR mode can invoke a PC routine that receives control in either AR mode or primary mode.

- **If the PC routine receives control in AR mode:**

  Parameter lists that the user passes can be located in the user's primary address space or any other address space except the user's secondary address space. An ALET must qualify any address that the user passes to the PC routine (An ALET identifies the address space that contains the passed address). An address passed in a general register must be qualified by an ALET in the AR that corresponds to the general register. If you are passing ALETs, you should be aware of the rules for passing ALETs, and how to check the validity of passed ALETs. For further information on passing ALETs, see “Special ALET Values at a Space Switch” on page 103, “Rules for passing ALETs” on page 108, and “Checking the authority of callers” on page 120.

  The user must not use ARs to pass anything except ALETs. For information on using ALETs, see Chapter 5, “Using access registers,” on page 93.

- **If the PC routines receives control in primary mode:**

  All addresses passed by the user's program must reside in the user's primary address space. **IBM recommends** that those addresses be ALET qualified. The value of the ALET must be 0.

Before issuing a PC instruction, the user must:

- Load the PC number into general register 14.
- If there is a parameter list to pass, load its address into general register 1. If the address is ALET qualified, load AR 1 with the ALET.

For more information about using ARs, see Chapter 5, “Using access registers,” on page 93.

PC routine that receives control in primary mode

After receiving control, the PC routine must establish a general register as a base register. The PC routine must also initialize general register 13:

- If the PC routine calls other routines, the PC routine must initialize general register 13 to the address of an 18-word save area that's located on a word boundary in the PC routine's primary address space. The PC routine must initialize the second word of the save area to the value C‘F1SA’. The value C‘F1SA’ indicates that the system saved the user's environment on the linkage stack. **IBM recommends** that all PC routines that receive control in primary mode initialize general register 13 in this way.
- A PC routine that does not call other routines and does not wish to provide an 18-word save area must initialize general register 13 to one of the following values.
  - Zero.
  - The address of a two word save area that's located on a word boundary in the PC routine's primary address space. The PC routine must initialize the second word of the area to the value C‘F1SA’.

  Either value, zero or C‘F1SA’, in general register 13 indicates that the system saved the user's environment on the linkage stack.

Addressability to the latent parameter area is through the primary address space. When the PC routine receives control, general register 4 contains the address of the latent parameter area.

Before returning control to the user, the PC routine must:

- Free any save area or work area it obtained.
- If there are parameters to pass, place their address into general register 0 or 1.
- If there is a return code, place it into general register 15.

To restore the user's environment and to return control, the PC routine must issue the PR instruction.

PC routine that receives control in AR mode

After receiving control, the PC routine must establish addressability by loading a base register. The PC routine must also load an ALET of 0 into the AR that corresponds to the base register.

Addresses that the caller passes to the PC routine must be qualified by an ALET. Before using an ALET, the PC routine must check the ALET:

- If the caller passes an ALET of 0, a space switch PC routine for which SASN=OLD has been specified must change the ALET to 1 before using it.
- If the caller passes other ALETs, the PC routine must use them to qualify addresses that the caller has passed.
- The PC routine must never use an ALET of 1 that the caller has passed. If a caller passes an ALET of 1, the PC routine might, for example, set an error return code and return to the caller.

Addressability to the latent parameter area is through the primary address space. When the PC routine receives control, general register 4 contains the address of the latent parameter area. Before referencing the latent parameter area, the PC routine must set AR 4 to a value of 0.

Before returning control to the user, the PC routine must do the following:

- If there are parameters to pass, place the address of the parameters into general register 0 or 1 and the ALET the caller will use to address the data into the corresponding AR. Remember that the address of any data in the caller's address space is qualified by an ALET of 0 for the caller, but an ALET of 1 for the PC routine if SASN=OLD. When passing the ALET to qualify the address of data in the caller's address space, **IBM recommends** that the PC routine pass an ALET of 0 rather than depending on the caller to change the ALET from 1 to 0.
- If there is a return code, place it into general register 15.

To restore the caller's environment and return control, the PC routine must issue the PR instruction.

The following examples compare the linkage conventions for the basic PC (first example) to the conventions for the stacking PC (second example). Both the user program and the PC routine are in primary mode.

User	PC Routine
⋮ * * BASIC PC LINKAGE * STM 14,12,12(13) ESAR 2 ST 2,16(,13) L 2,PCNUMBER PC 0(2) L 2,16(,13) SSAR 2 LM 2,12,28(13) ⋮	BALR 6,0 PCLINK STACK ⋮ PCLINK UNSTACK PT 3,14

User	PC Routine
. . * * STACKING PC LINKAGE * L 14,PCNUMBER PC 0(14) .	BALR 6,0 ⋮ PR

If the PC routine is in AR mode, the following is an example of the instructions you can use to establish addressability:

```
BALR   6,0
USING  *,6
SLR    7,7
SAR    6,7
```

See Chapter 5, “Using access registers,” on page 93 for information about being in AR mode and manipulating the contents of ARs.

Resource Management

IBM recommends that a program running under the job step task, rather than under a subtask of the job step task, acquire and release these cross memory resources: AXs, EAXs, LXs, authority tables, and entry tables. Likewise, the same program should load the PC routines.

During normal termination, the program that obtained cross memory resources should release those resources. If this is not done, however, MVS releases these resources during termination of the job step task.

When the job step task of an address space terminates, MVS eliminates any cross memory connections between the terminating address space and other address spaces. After these connections are eliminated:

- Programs executing in other address spaces cannot access the terminating address space through a PT, SSAR, or PC instruction.
- Programs executing in other address spaces cannot use ARs to access the terminating address space.
- Subsequent job steps can execute but cannot obtain cross memory resources. If such a job step issues an LXRES, AXRES, or ETCRE macro, the system returns an X'052' abend code.

Reusing ASIDs

The system assigns an ASID to an address space when the address space is created. A limited number of ASIDs are available for the system to assign. When all ASIDs are assigned to existing address spaces, the system is unable to start a new address space. This condition might be the result of too many **lost ASIDs** in the system. A lost ASID is one that is associated with an address space that has terminated, but because of the address space's cross memory connections, the system does not reuse the ASID. In effect, the ASID is "lost from use" for the duration of the IPL, or until all connected address spaces have terminated.

This section tells you two ways to reduce the possibility that the system will run out of ASIDs for assignment to new address spaces. One is through coding cross memory services to avoid losing ASIDs and the second is through the installation's use of parameters in the IEASYSxx member of SYS1.PARMLIB.

Coding cross memory services to avoid the loss of ASIDs from reuse

A reusable ASID is one which has at some time been assigned to an address space created by a START command which specified REUSASID=YES, or an ASCRE macro which specified ATTR=REUSASID. Other ASIDs are referred to as ordinary ASIDs. All ASIDs are initially ordinary ASIDs. If no reusable ASIDs are available when a START command or ASCRE requests a reusable ASID, the system converts an available ordinary ASID into a reusable ASID. Once converted, an available reusable ASID can be used only to satisfy a request for a reusable ASID. The system honors a request for a reusable ASID only if REUSASID(YES) is specified in parmlib member DIAGxx. Otherwise, the system assigns an ordinary ASID. This allows the installation to enable reusable ASIDs only after testing its product set to verify that any maintenance required to support reusable ASIDs has been installed.

As you code cross memory services, try whenever possible to allow the ASID of an address space to be free for reuse at address space termination. To do this, you need to know the circumstances under which the system does not reuse an ordinary ASID. When an ordinary address space terminates, the system considers reusing the ASID that is associated with that address space. Whether the ordinary ASID is available for reuse depends on the cross memory connections that have been established between that address space and other address spaces.

An ordinary ASID is unavailable for reuse when the address space owns entry tables that contain space switch entries (created through SSWITCH=YES on the ETDEF macro), and when one of the following is true:

- Those tables connect to other address spaces through a **non-system LX** (created through SYSTEM=NO on the LXRES macro), in which case the ASID is not eligible for reuse until all connected address spaces terminate
- Those tables connect to other address spaces through a **system LX**, in which case the ASID is not eligible for reuse for the duration of the IPL.

The ASID of an ordinary address space with no entry tables, or with entry tables that contain only non-space switch entries (created through SSWITCH=NO on the ETDEF macro), is available for reuse when the address space terminates. A reusable ASID is available for reuse when the address space terminates (but only by a subsequent request for a reusable ASID) regardless of non-system LX or system LX connections.

For an example of ordinary ASID reuse, see Figure 17 on page 54, which describes the cross memory relationships between four ordinary address spaces. Address spaces A, B, and C own entry tables with

space switch entries. Address space B is a server address space. It has a system LX; its PC routines are available to all address spaces. Address spaces A and C have non-system LXs; their PC routines are available to selected address spaces. Address space D owns no entry tables.

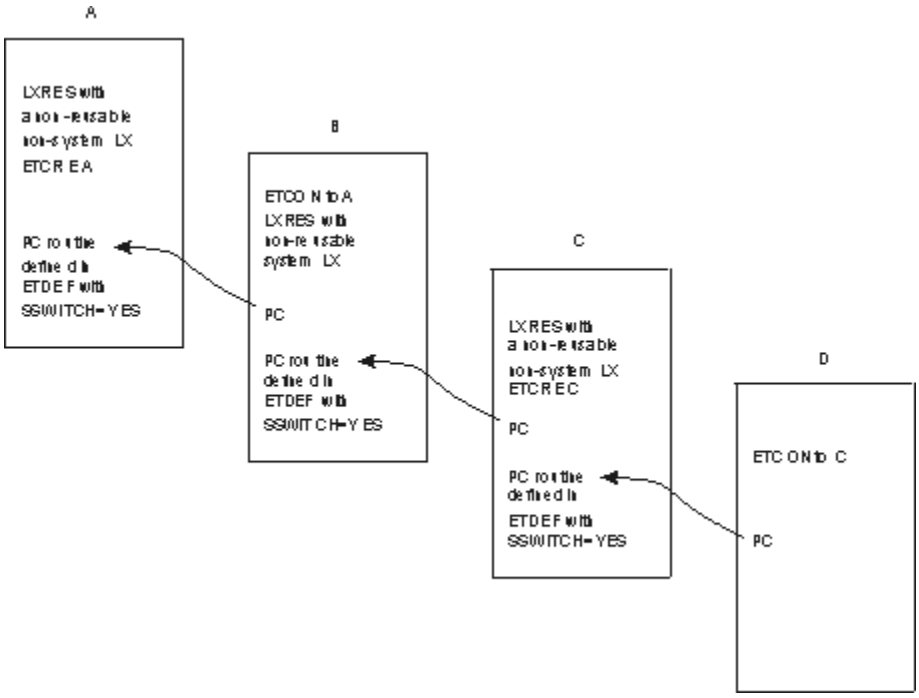


Figure 17. Cross memory connections between address spaces

To maintain the integrity of an address space, the system does not reuse an ASID until all programs that could potentially access that address space have completed. This means that the system reuses the ASIDs of the address spaces in the figure as follows:

- A's and B's ASIDs are reusable only after a reIPL.
- C's ASID is reusable after both C and D terminate.
- D's ASID is reusable after D terminates.

A's and B's ASIDs are the lost ASIDs. Because programs in all address spaces potentially have the ability to transfer control to address space B, and programs in B can transfer control to address space A, A's and B's ASIDs are not reusable within an IPL. (Consider the consequences of the system reusing A's ASID at termination of A. Then, a program in B could pass control to code running in the address space that received the reused ASID.)

Connecting an entry table with space-switch entries through a non-system LX to a system address space or a long-running address space (such as VTAM®, CICS®, DB2®, or JES) makes the ordinary ASID of the owner of the entry table non-reusable. Therefore, to avoid unnecessary loss of ASIDs, **IBM recommends** that you follow these rules:

- Use system LXs only when the cross memory service is to be used by all address spaces and the cross memory service provider is a long-running address space. If possible, use a reusable ASID for the cross memory service provider when using a system LX.
- Avoid connecting non-system LXs to long-running address spaces, or if possible, use reusable ASIDs for address spaces which own non-system LXs which are connected to long-running address spaces.

Coding to allow use of reusable ASIDs

Look for instances in your program where any of the following macros are specified in cross-memory mode (that is, when Primary ASID (PASID) ^= Secondary ASID (SASID)):

ATSET	CPUTIMER	GQSCAN
AXEXT	DIV	LXFRE
AXFRE	ETCON	LXRES
AXRES	ETCRE	SYMREC
AXSET	ETDES	VSMLIST
CPOOL	ETDIS	VSMLOC

Prior to z/OS 1.6, these services used a basic PC, and the macros generated a SSAR to restore SASID upon return from the PC. If the macro is issued when PASID ^= SASID and SASID is a reusable ASID, the SSAR causes a program check code X'0013', which results in a 0D3 abend.

In z/OS 1.6, these services were changed to use a stacking PC, so the SSAR generated by the macros is unnecessary when the program executes on z/OS 1.6, or later release. The macros avoid generating the SSAR if SYSSTATE OSREL=ZOSV1R6 has been used to indicate that the program can assume it is running on a z/OS1.6 or later level system. If the program could run on earlier levels, then it is necessary to generate two expansions of the macro—one with SYSSTATE OSREL=ZOSV1R6 , and one without, and then test CVTH7709 at execution time and execute the appropriate expansion.

Note: GQSCAN was changed to a stacking PC in OS/390® V1R2, but the GQSCAN macro continued to generate a SSAR so that the expansion would be compatible with earlier releases. In z/OS 1.6, the GQSCAN macro was changed to avoid generating SSAR regardless of SYSSTATE OSREL, since the SSAR was required only when executing on a release earlier than OS/390 V1R2.

Look for instances in your program where you code a SSAR instruction. If PASID ^= the target SASID and the target SASID is a reusable ASID, the SSAR gets a program check code X'0013', which results in a X'0D3' abend. You must obtain the instance number of the intended target address space (the EPAIR, ESAIR, or ESTA code 5 instructions may be useful for this), and use SSAIR instead of SSAR. Note that your program might be executing on a processor or old level (pre-1.6) level of z/OS where these instructions are not available. CVTALR or PSAALR should be tested to see if the new instructions are available. Continue to use SSAR when CVTALR or PSAALR is off. If you know that your program will be running on z/OS 2.1 or later, you do not need to check CVTALR or PSAALR because those bits will always be on for those releases.

Look for instances in your program where you code a PT instruction, where the target PASID is not the current PASID. In most cases, this would be a PT to return to the issuer of a space-switching basic PC. For these cases, the preferred solution is to convert to a stacking PC, and return via PR instead of PT. This solution can be used regardless of the setting of CVTALR and PSAALR. For other uses of PT, if CVTALR or PSAALR is on, obtain the instance number of the intended target address space at an appropriate time, and use PTI instead of PT.

Make sure that your program does not cause X'0D3' abends when other products exploit reusable ASIDs. Look at code that can be invoked in other address spaces. Be especially aware of code that runs in all address spaces, such as a task termination resource manager established through RESMGR TYPE=TASK,ASID=ALL or an SSI EOT function routine.

When to use a reusable ASID

Consider the following situations when determining when to use a reusable ASID:

- If you have an address space that becomes nonreusable when it terminates, as identified by the message IEF352I ADDRESS SPACE UNAVAILABLE at termination, consider using a reusable ASID with it.
- If you have an address space that does not become nonreusable when it terminates, do not use a reusable ASID with it.

To use a reusable ASID, specify REUSASID=YES on the START command for the address space or specify ATTR=(REUSASID) on the ASCRE.

Note: If there is a request for a reusable ASID but there are no available ASIDs that were previously used as reusable ASIDs, z/OS takes an ASID from the pool of available ordinary ASIDs to satisfy the request.

If any 0D3 abends then result from the use of a reusable ASID, investigate and resolve them.

Attention: As soon as an ASID is used to satisfy a request for a reusable ASID, it must always serve as a reusable ASID; you cannot then use it for a START without REUSASID=YES or an ASCRE without ATTR=(REUSASID). Unnecessary use of REUSASID=YES or ATTR=(REUSASID) can reduce the number of ordinary ASIDs that are available for satisfying ordinary address space requests.

Using IEASYSxx to Avoid Running Out of ASIDs

A second way you can reduce the possibility that the system will run out of ASIDs is to reserve ASIDs through the RSVNONR and RSVSTRT parameters in the IEASYSxx member of SYS1.PARMLIB. The reserved ASIDs replace those lost due to cross memory activity. See *z/OS MVS Initialization and Tuning Reference* for more information about specifying those parameters.

Reassigning LXs when the LX reuse facility is enabled

The limit on the number of LXs is 32768. Some of the LXs are reserved as system LXs; the rest are available as non-system LXs. You can use the NSYSLX parameter in the IEASYSxx member of SYS1.PARMLIB to set the number of system LXs available for the system's use. System and non-system LXs are subdivided into 12-bit LXs and 24-bit LXs. 12-bit LXs are in the range 0-2047, and 24-bit LXs are in the range 2048-32768 (the specific value of the LX depends upon the LXSIZE parameter of the LXRES macro).

Use the LXRES macro with SYSTEM=YES to obtain a system LX; with SYSTEM=NO to obtain a non-system LX. Use the LXRES macro with LXSIZE=12 to obtain a 12-bit LX; with LXSIZE=16|23|24 to obtain a 24-bit LX. 24-bit LXs are further subdivided into reusable and non-reusable LXs.

Guideline: Always request a 24-bit LX when running on z/OS V1R6 or later.

There are different rules for each category of LX.

Non-Reusable system LX
The system does not reassign the LX. The original requestor of the LX should reconnect to the LX if the address space terminates and then restarts.

Reusable system LX
The system reassigns the LX after the owning address space has terminated or after the owner has used LXFRE to free the LX.

Non-Reusable non-system LX
The system reassigns the LX after all entry tables are disconnected from the LX. This is the case when the owning address space terminates or when the owner uses ETDIS to disconnect the entry table from the LX and uses LXFRE to free the LX; and all address spaces that have used ETCON to connect the LX to that entry table have either terminated or used ETDIS to disconnect the entry table from the LX.

Reusable non-system LX
The system reassigns the LX after the owning address space has terminated or after the owner has used LXFRE to free the LX.

Guideline: IBM suggests that when using non-system LXs you use reusable non-system LXs.

Reassigning LXs when the LX Reuse Facility is not enabled:

The limit on the number of LXs is 2048. Some of the LXs are reserved as system LXs; the rest are available as non-system LXs. You can use the NSYSLX parameter in the IEASYSxx member of SYS1.PARMLIB to set the number of system LXs available for the system's use. Use the LXRES macro with SYSTEM=YES to obtain a system LX; with SYSTEM=NO to obtain a non-system LX. The rule for the reuse of a system LX is simple: the system does not reassign it. The original requester of the LX can choose to reconnect to the LX should the address space terminate and then restart. The system considers reusing a non-system LX when all entry tables are disconnected from the LX. This is the case when:

- an address space that owns a non-system LX terminates or when the owner of the non-system LX uses ETDIS to disconnect all entry tables the entry table from the LX and uses LXFRE to free the LX; and

- all address spaces that have used ETCON to connect the LX to that entry table have either terminated or used ETDIS to disconnect the entry table from the LX.

Example of Reassigning LXs

In the example in Figure 17 on page 54, assume that all entry tables are disconnected by the system during address space termination. This means the system reassigns non-system LXs as follows:

- A's non-system LX is reassignable when
  - A terminates or issues LXFRE; and
  - B terminates or issues ETDIS to disconnect from A's LX. If this had been a reusable non-system LX, it would become reassignable regardless of B's action.
- C's non-system LX is reassignable when
  - C terminates or issues LXFRE; and
  - D terminates or issues ETDIS to disconnect from C's LX. If this had been a reusable non-system LX, it would become reassignable regardless of D's action.

Reusing AXs and EAXs

The combined number of AXs and EAXs available for all programs in the system is 16382. When an address space that owns AXs terminates or when the AXs are explicitly freed through AXFRE, those AXs are available for the system to reassign.

The system reuses EAXs in the same way it reuses ASIDs. The system does not reuse an EAX until all programs that could potentially access the address space have completed. In the example in Figure 17 on page 54, assume that A owns an EAX in the entry table connected to B, and B owns an EAX in an entry table connected to all address spaces (because B has a system LX), and C owns an EAX in the entry table connected to D. In this example, the system reuses the EAXs as follows:

- A's EAX is reusable only after a reIPL (because A connected to B, which owns a system LX).
- B's EAX is reusable only after a reIPL (because B owns a system LX).
- C's EAX is reusable after both C and D terminate.

PC Routine Loading Recommendations

MVS deletes PC routines when the task that loaded the PC routine terminates. Therefore, **IBM recommends** that PC routines be loaded by a program running under the job step task of the address space that creates and owns the PC entry tables. If the program that loads the PC routine is running under a task that's subordinate to the job step task and the subordinate task terminates, the PC routine will be deleted from virtual storage. Cross memory connections to that PC routine remain, however, until the job step task terminates. If a program issues a PC instruction to invoke the deleted PC routine, the results will be unpredictable: a program interrupt may occur or other random errors may occur if the virtual storage previously occupied by the deleted program has been reused.

Accounting Considerations

CPU execution time for space switch PC routines is attributed to the home address space of the work unit that invokes the PC routine. The PC routine execution time is not attributed to the address space where the PC routine itself resides. For example, address space A owns a space switching PC routine that is invoked by a task whose home address space is B. When the task in B executes the PC routine in space A, that CPU time is attributed to address space B.

Recovery Considerations

Space switch PC routines have special recovery considerations. A space switch PC routine has active binds to address spaces other than home. If the PC routine tries to access data in one of these address spaces after the address space has terminated, the PC routine will incur a program check and its recovery

routine might get control. The SETFRR macro provides options that specify the cross memory mode in which the recovery routine must get control. The ETDEF macro with the ARR parameter and the ESTAEX macro also can define recovery routines for PC routines in cross memory mode. However, these recovery routines are not protected against memory terminations of associated address spaces.

There are also options that enable a recovery routine to get control as a resource manager when the requested cross memory mode cannot be established in order to recover resources serialized by local (CML) or global locks. For information on recovery in cross memory mode, see *z/OS MVS Programming: Authorized Assembler Services Guide*.

Chapter 4. Using the 64-bit address space

This chapter describes how to use the address space virtual storage above 2 gigabytes and control the physical frames that back this storage.

What is the 64-bit address space?

Because of changes in the architecture that supports the Multiple Virtual Storage (MVS) operating system, there have been two different address spaces prior to the 64-bit address space. The address space of the 1970s began at address 0 and ended at 16 megabytes. The architecture that created this address space provided 24-bit addresses.

In the early 1980s, XA (extended architecture) introduced an address space that began at address 0 and ended at two gigabytes. The architecture that created this address space provided 31-bit addresses. To maintain program compatibility, MVS provided two addressing modes (AMODEs):

- Programs that run in AMODE 24 can use only the first 16 megabytes of the address space
- Programs that run in AMODE 31 can use the entire 2 gigabytes.

Today, the address space begins at address 0 and ends at 16 exabytes. The architecture that creates this address space provides 64-bit addresses. The address space structure below the 2 gigabyte address has not changed; all programs in AMODE 24 and AMODE 31 continue to run without change. In some fundamental ways, the address space is much the same as the XA address space.

In the 31-bit address space, a virtual **line** marks the 16-megabyte address. The 64-bit address space also includes the virtual line at the 16-megabyte address; additionally, it includes a second virtual line called **the bar** that marks the 2-gigabyte address. The bar separates storage below the 2-gigabyte address, called **below the bar**, from storage above the 2-gigabyte address, called **above the bar**. The area above the bar is intended for data; no programs run above the bar. IBM reserves an area of storage above the bar for special uses to be developed in the future.

You can set a limit on how much virtual storage above the bar each address space can use. This limit is called the **MEMLIMIT**. If you do not set a MEMLIMIT, the system default is 2G, meaning that the address space can use up to 2G of virtual storage above the bar. If you want an address space to have access to more or less virtual storage above the bar, you must explicitly set the MEMLIMIT to the limit you want. You can set an installation default MEMLIMIT through System Management Facility (SMF). You can also set a MEMLIMIT for a specific address space in the job control language (JCL) that creates the address space or by using SMF exit IEFUSI or the SMFLIMxx parmlib member. For information about how to set MEMLIMIT explicitly, see “Limiting the use of private memory objects” on page 62.

Figure 18 on page 60 shows a z/OS address space, including the line that marks the 16-megabyte address, the bar that marks the 2-gigabyte address, and the default shared area starting at 2 terabytes and ending at 512 terabytes.

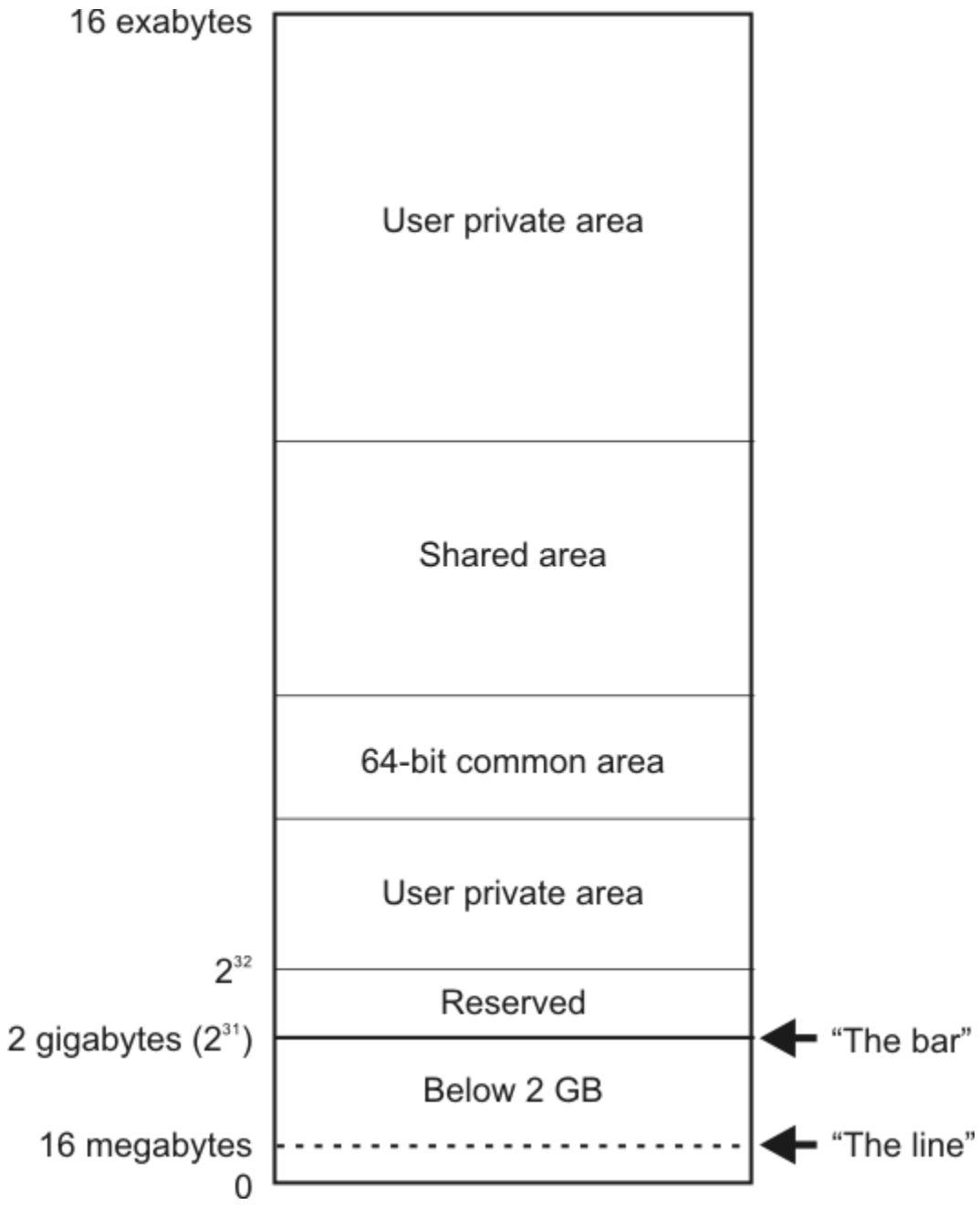


Figure 18. z/OS address space

Before z/OS Version 1 Release 3, all programs in AMODE 31 or AMODE 24 were unable to work with data above the bar. To use virtual storage above the bar, a program must request storage above the bar, be in AMODE 64, and use the z/Architecture® assembler instructions.

As of z/OS Version 1 Release 5, the following enhancements for 64-bit virtual storage have been added:

- 64-bit shared memory support
- Multiple guard area support for private high virtual storage
- Default shared memory addressing area between 2 terabytes and 512 terabytes

As of z/OS Version 1 Release 10, support for 64-bit common virtual storage has been added. The size of the 64-bit common area can be specified using the HVCOMMON keyword in the IEASYS*xx* member of parmlib or during system IPL in response to the IEA101A message. The 64-bit common area will reside on a 2-gigabyte boundary and the total size will be a multiple of 2 GB. The minimum size is 2 GB, the maximum is 1 TB, and the default is 64 GB.

Why would you use virtual storage above the bar?

The reason why someone designing an application would want to use the area above the bar is simple: the program needs more virtual storage than the first 2-gigabyte address space provides. Prior to z/OS Version 1 Release 2, a program's need for storage beyond what the former 2-gigabyte address space provided was sometimes met by creating one or more data spaces or hiperspaces and then designing a memory management schema to keep track of the data in those spaces. Sometimes programs written before z/OS Version 1 Release 2 used complex algorithms to manage storage, reallocate and reuse areas, and check storage availability. With the 16-exabyte address space, these kinds of programming complexities are unnecessary. (An exabyte is slightly more than one billion gigabytes.) A program can potentially have as much virtual storage as it needs, while containing the data within the program's primary or home address space.

A good example of a programming model that can successfully take advantage of the 16-exabyte address space is a program that needs very large buffer pools. This program has typically used multiple data spaces and then managed them separately and uniquely. With the 16-exabyte address space, a program can use the area above two gigabytes for a buffer pool. A simple memory mapping scheme is all that is needed to keep track of the data.

Memory management above the bar

Virtual memory above 2 GB is organized as memory objects that a program creates. A memory object is a contiguous range of virtual addresses that are allocated by programs as a number of application pages which are 1-megabyte multiples on a 1 MB boundary. Programs continue to run and execute in the first 2 GB of the address space.

Memory objects

Programs obtain storage above the bar in "chunks" of virtual storage called **memory objects**. The system allocates a memory object as a number of virtual segments; each segment is a megabyte in size and begins on a megabyte boundary. A memory object can be as large as the memory limits set by your installation and as small as 1 megabyte. Other attributes of a memory object include the following characteristics:

- The storage key is defined by the program; for an unauthorized program, the storage key at the time of issuing the IARV64 macro is the program's PSW key.
- You can specify whether or not you want the memory object to be fetch protected. There is no change key support for virtual storage above the bar.
- The owner of a private memory object is the TCB of the program that creates the private memory object, or a TCB to which the creating program assigns ownership. If an SRB creates a private memory object, the SRB must assign ownership of the private memory object to a task.
- A shared memory object is system-owned. The cross-memory resource owner (CMRO) TCB of the address space owns the shared interest in the shared memory object.
- A common memory object is visible at the same address in *every* address space and, once it is created, every address space has access to a common memory object without having to explicitly request access. An owner is associated with a common memory object for diagnostic purposes.

Using the IARV64 macro, a program can create and free a memory object and manage the physical frames that back the virtual storage. You can think of IARV64 as the GETMAIN/FREEMAIN or STORAGE macro for virtual storage above the bar. (GETMAIN/FREEMAIN and STORAGE do not work on virtual storage above the bar.)

When a program creates a memory object, it provides an area in which the system returns the memory object's low address. You can think of the address as the name of the memory object. After creating the memory object, the program can use the storage in the memory object as it used storage in the 2-gigabyte address space; see “Using a memory object” on page 71. The program cannot safely operate on storage areas that span more than one memory object.

An authorized program can ask the system to pagefix areas of private memory objects, making pages unavailable for stealing. The program specifies the ranges of pages that the system is to fix. Later, the program can undo the pagefix operation.

An authorized program cannot pagefix 64-bit shared memory objects. For more information about shared memory objects, see "Creating shared memory objects" on page 79.

To help the system manage the physical pages that back ranges of addresses in memory objects, a program can alert the system to its use of some of those pages by issuing an IARV64 PAGEOUT request thereby making them available for the system to steal.

The program can free the physical pages that back ranges of memory objects and, optionally, clear those ranges to zeros. Later, the program can ask the system to return the physical backing from auxiliary storage. When it no longer needs the memory object, the program frees it in its entirety.

The program can identify one user token for both private and shared memory objects. Then, if a IARV64 DETACH AFFINITY=LOCAL request is issued for that user token, the private memory objects will be detached and destroyed and the interest in the shared memory objects specified by the user token will be removed from the shared memory object.

Limiting the use of private memory objects

While there is no practical limit to the virtual storage above the bar, practical limits exist to the finite real storage frames and auxiliary storage slots that back the virtual storage. Limiting the virtual storage that an address space can consume will directly limit its real frame and auxiliary storage usage. In addition to limiting the virtual storage usage to control real frame usage, your installation can limit an address space or set of address space's real frame usage by classifying the address spaces to a WLM resource group with a real storage memory limit. However, the amount of auxiliary storage cannot be limited by resource group. For more information about WLM resource groups, see *z/OS MVS Initialization and Tuning Guide* and *z/OS MVS Planning: Workload Management*.

Conceptually, the aggregate of the unlimited virtual storage for all concurrently active address spaces must fit into the amount of real storage and auxiliary storage defined on a given system. The fixed virtual storage, which is not pageable, must fit in the available real storage, and anything else can potentially be paged out to auxiliary storage. However, WLM controls system resources such that when resources are limited, WLM attempts to control address space consumption by taking actions such as altering dispatch priorities, swapping out address spaces, and so on. Even so, there are conditions under which the system may run out of these resources (such as when real storage is constrained) and require the system to start swapping or paging, which can unacceptably impact system performance. For more information about real frame and auxiliary storage resources and how to determine your required auxiliary and real storage requirements, see Auxiliary storage management initialization in *z/OS MVS Initialization and Tuning Guide*.

The amount of virtual storage that an address space can use for private memory objects at any given time is controlled by the MEMLIMIT (memory limit) specification. However, note that authorized users of IARV64 GETSTOR who specify the MEMLIMIT=NO or MEMLIMIT=COND parameters may request that the storage obtained for that specific GETSTOR request not be subject to the MEMLIMIT; they may use other means of determining whether or not the storage associated with the allocation will have an impact on the system. The MEMLIMIT for an address space can be set via the IEFUSI installation exit, an SMFLIMxx parmlib member specification, a JCL specification, the system's default MEMLIMIT (specified in the SMFPRMxx parmlib member), and other means if under the control of z/OS UNIX Systems Services (z/OS UNIX).

It is recommended that you review the requirements for your applications and use those requirements as a guide to setting the MEMLIMIT for those application address spaces. In many cases, JCL, SMFLIMxx, or z/OS UNIX (also known as OMVS) controls would be recommended based on the expected workload demand and configuration. For all other ad hoc types of address spaces, the system-wide default would most likely apply.

z/OS UNIX address spaces have additional MEMLIMIT controls (such as the MEMLIMIT parameter in the RACF OMVS segment, the SETOMVS MEMLIMIT command, and others) that control dubbed address spaces that are also under its control. The default MEMLMIT specified in the SMFPRMxx parmlib member

may still apply to those address spaces. Before proceeding, see *z/OS UNIX System Services Planning* for more information.

After an address space is started, its MEMLIMIT can only be changed if its MEMLIMIT is currently assigned via the SMF default or it is a dubbed z/OS UNIX address space.

Determining the effective MEMLIMIT value

Outside of z/OS UNIX controls that can take precedence over all other controls, the MEMLIMIT that is used at the start of a job step is based on the following conditions, in order of precedence, as shown in Figure 19 on page 64. The first condition that is satisfied determines the MEMLIMIT value. (The numbered blocks in the figure refer to the numbered steps in the detailed list that immediately follows the figure.)

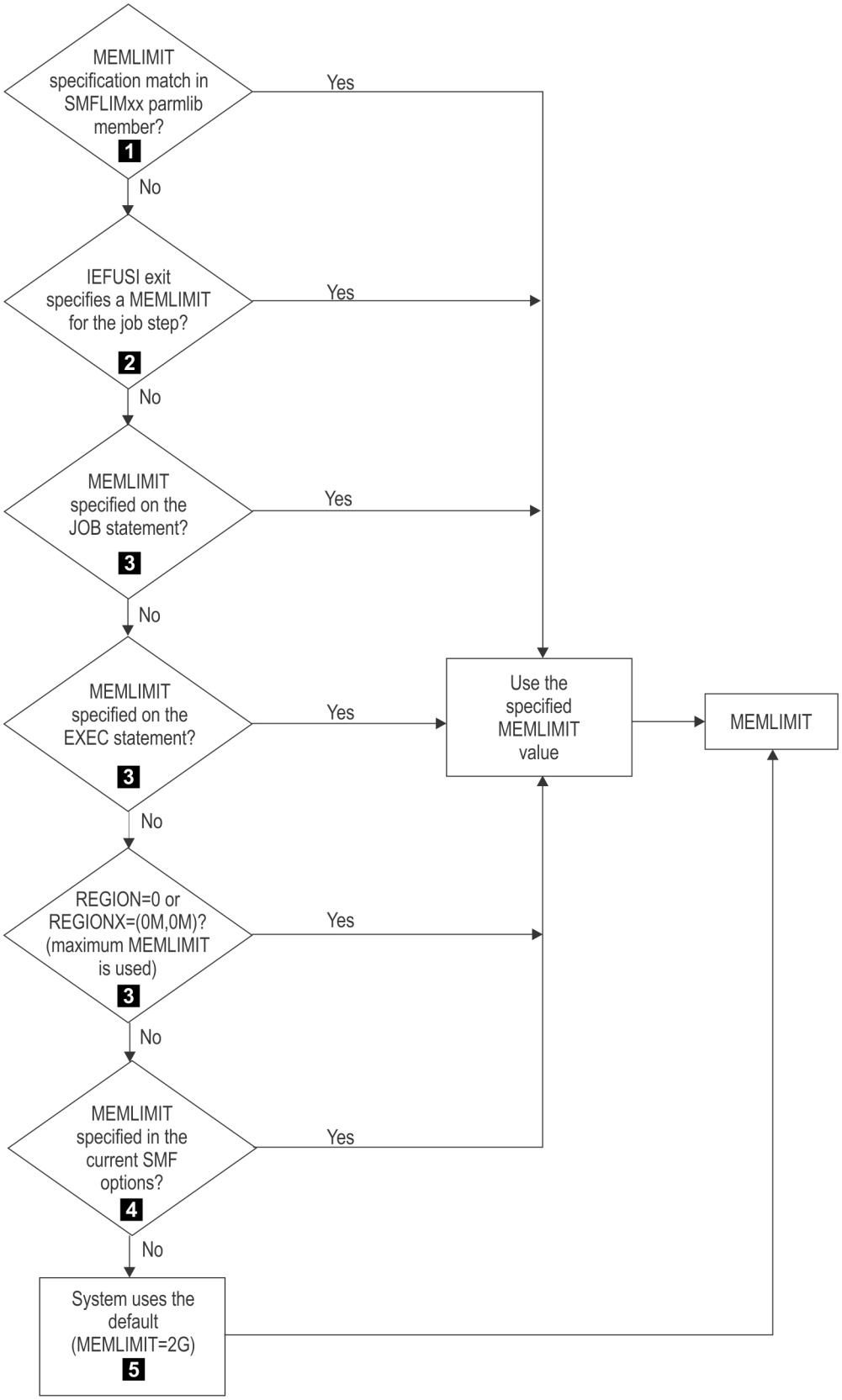


Figure 19. Order of precedence in determining the effective MEMLIMIT value

The following sequence describes the order of precedence, as shown in Figure 19 on page 64, for determining the MEMLIMIT value to apply:

1. A matching MEMLIMIT specification is found in the SMFLIMxx parmlib member.
   - SMFLIMxx provides a broad set of filter keywords with wild card capabilities, such as JOBNAME, JOBCLASS, USER, SUBSYS, and PGMNAME.

     Note: It is possible to use SMFLIMxx REGION MEMLIMIT(*nn*) to set a MEMLIMIT for all job steps that do not contain a more specific SMFLIMxx filter specification.
   - The NOHONORIEFUSIREGION attribute in the program property table (PPT) for the job prevents SMFLIMxx from altering the MEMLIMIT.
   - See Using SMFLIMxx to control the REGION and MEMLIMIT in *z/OS MVS Initialization and Tuning Guide* for more information.
2. A specific MEMLIMIT for the job is specified via the IEFUSI exit.
   - The IEFUSI exit can override SMFLIMxx, if requested.
   - The NOHONORIEFUSIREGION attribute in the program property table (PPT) for the job prevents IEFUSI from altering the MEMLIMIT.
   - See Using exit routines to limit region size in *z/OS MVS Initialization and Tuning Guide*, and IEFUSI — Step initiation exit in *z/OS MVS Installation Exits* for more information.
3. The MEMLIMIT is determined from JOB or EXEC statement specifications.
   - When the MEMLIMIT parameter is specified, it is used. Specifying MEMLIMIT on the JOB statement overrides MEMLIMIT on the EXEC statement.
   - When REGION=0K, REGION=0M, or REGIONX=(0M,0M) is specified, the MEMLIMIT is set to NOLIMIT.
   - If REGION=0K, REGION=0M, or REGIONX=(0M,0M) is specified and the IEFUSI exit or SMFLIMxx member limits the REGION size but does not set a MEMLIMIT, MEMLIMIT is defaulted to the REGION size above 16 MB.
   - See *z/OS MVS JCL Reference* for more information.
4. A system-wide default SMF MEMLIMIT value has been specified in the current SMF option.
   - The SETSMF command can be used to change the current default SMF MEMLIMIT value from the one specified in the active SMFPRMxx parmlib member.

     By contrast, the SET SMF command can be used to change the active SMFPRMxx parmlib member, which may specify a new default SMF MEMLIMIT value.

     You can use the D SMF,O command to display the SMF options and values that are currently in effect.
   - The SETSMF and SET SMF commands temporarily change the options that are in use on the system on which they are issued; the changes are lost upon the next IPL of the system. To make the changes permanent, update the SMFPRMxx parmlib member that is used during IPL.
   - See "Setting the system-wide default SMF MEMLIMIT value" on page 65, "Altering the default SMF MEMLIMIT value" on page 66, and SMFPRMxx in *z/OS MVS Initialization and Tuning Reference* for more information about using the SET SMF and SETSMF commands.
5. If no default SMF MEMLIMIT parameter has been specified in the active SMFPRMxx parmlib member and no SETSMF command has been issued to specify a default SMF MEMLIMIT value, the system uses a MEMLIMIT value of 2G.

Setting the system-wide default SMF MEMLIMIT value

You can set the current system-wide SMF MEMLIMIT default in the following ways:

- The MEMLIMITparameter in the active SMFPRMxx parmlib member, where the active SMFPRMxx member was established either at IPL time or changed via the SET SMF=xx command.
- The SETSMF MEMLIMIT(*nn*) command on a specific system or systems without changing the active SMFPRMxx member.

The default SMF MEMLIMIT does not apply to all address spaces; thus, you cannot use the SET SMF or SETSMF commands to change an address space's MEMLIMIT value that was last set by any means other than the default SMF MEMLIMIT.

Note that a MEMLIMIT value can be set for all job steps through a generic REGION MEMLIMIT(xx) specification in the active SMFLIMxx parmlib member. However, for the purpose of this documentation, this is not considered the SMF default; rather, it is an SMFLIMxx specification match, which differs in the following ways:

- An SMFLIMxx specification always takes precedence over the SMF default value.
- An SMFLIMxx specification only takes effect at the start of the next job step, whereas the SMF default takes effect immediately even for currently running job steps.

Altering the default SMF MEMLIMIT value

Before altering the default SMF MEMLIMIT, keep in mind that only address spaces that are currently using the default SMF MEMLIMIT or will use it on the next JOB step start will be affected.

- Before increasing the default SMF MEMLIMIT, you must consider how it might impact real and auxiliary resources, especially when many address spaces might be affected. How applications manage their memory restrictions can also have unexpected, and perhaps unpredictable, impact. Some application behavior conditions to consider are:
  - An application might never approach its MEMLIMIT, so increasing the MEMLIMIT would have no effect on that application.
  - An application might conditionally allocate virtual storage based on its MEMLIMIT at the start of a job step and, thus, might allocate more storage at the start of the next job step.
  - An application might attempt to allocate virtual storage on a periodic or demand basis and, thus, would increase its memory usage at some future time or without starting a new job step.
  - An application might allocate more virtual storage if it detects a change in its MEMLIMIT.
- Before decreasing the default SMF MEMLIMIT, consider the following points:
  - The effect that the change may have on job steps that previously allocated or required more 64-bit high virtual storage than the new lower MEMLIMIT.
  - The dynamic affect that the change will have on currently running job steps that are were assigned the default SMF MEMLIMIT.
    - Their effective MEMLIMIT is always the maximum of what the last job step started with and the current MEMLIMIT. As such, their highest previous allocation amount is not taken into account.
    - Their current allocation may be higher than their effective MEMLIMIT. In such cases, they continue to run, but they cannot allocate any 64-bit high virtual storage because they are beyond their limit.

The MEMLIMIT for job steps that use the SMF system default limit is always the higher of the current SMF system default MEMLIMIT or the SMF default MEMLIMIT at job step start. Thus, if a SET SMF or SETSMF command changes the default MEMLIMIT for an address space that is already created, the following changes occur:

- If the command increases the current default MEMLIMIT, all address spaces whose MEMLIMIT values are set through SMF run with the new, higher default.
- If the command decreases the current default MEMLIMIT, all address spaces whose MEMLIMIT values are set through SMF use the higher of the new (lower) default MEMLIMIT and their original (higher) system default that was assigned at job step start time.

  Lowering the default MEMLIMIT makes it possible for address spaces to have more allocated virtual storage than their current MEMLIMIT. Their deallocation of high virtual storage will bring their allocated storage down, but new allocations are restricted by their current MEMLIMIT and not the previously attained highest allocation level that they achieved at any time in the past.

MEMLIMIT and the 64-bit high virtual services

The system enforces the MEMLIMIT on the IARV64 REQUEST=GETSTOR, IARV64 REQUEST=CHANGEGUARD, and any other 64-bit high virtual services that allocate virtual storage. Note that guard areas and memory objects allocated by authorized programs that specify the MEMLIMIT=NO attribute are not subject to MEMLIMIT restrictions. As guard areas are never backed, they do not consume real frames or auxiliary storage, but MEMLIMIT=NO areas do. The SMF 30 field SMF30HVH provides the high-water mark for high virtual allocations for a given address, which includes MEMLIMIT=NO and MEMLIMIT=YES but not the GUARDAREAS. When an unconditional request for new storage (either for a new memory object or for more usable storage in an existing memory object) causes the MEMLIMIT to be exceeded, the system abends the program. IBM recommends that programs use the COND parameter to make a conditional request and check the return code to ensure that the storage is available.

Using SMF type 30 records to track real and auxiliary storage usage

SMF type 30 (common address space work) records provide several fields related to 64-bit, high virtual private storage, real frame, and auxiliary storage usage. Note that SMF30HVH, the high-water value for the number of 64-bit high virtual bytes, includes MEMLIMIT=NO and MEMLIMIT=YES allocated storage but not guard areas, so it is useful for noting potential real and auxiliary storage usage. However, since it includes MEMLIMIT=NO allocations, it is not exactly what is compared to the MEMLIMIT.

Using large pages

A *large page* has a page size larger than 4K, such as a 1 MB or 2 GB page, and is a special-purpose performance feature for memory objects. 1 MB pages can be pageable or fixed, but 2 GB pages are always fixed.

To request large pages for backing a memory object, authorized programs and unauthorized programs with read authority to the IARRSM.LRGPAGES resource in the FACILITY class can specify the PAGEFRAMESIZE parameter when issuing the IARV64 GETSTOR request. Programs can choose to have the system acquire and permanently fix 1 MB pages when allocated or dynamically fix and unfix them by using the IARV64 PAGEFIX and PAGEUNFIX requests. For pageable 1 MB pages, the system uses 4K pages when a page is referenced and there is not enough contiguous storage to back the page with a 1 MB page. Authorized programs can also request large pages for common memory objects by using the PAGEFRAMESIZE parameter when issuing the IARV64 GETCOMMON request. 1 MB pageable pages can also be used to back dataspace storage by using the PAGEFRAMESIZE parameter on the DSPSERV service.

The system programmer should carefully consider the following factors while determining which applications are to be granted access to large pages:

- Memory usage
- Page translation overhead for the workload
- Availability of fixed 1 MB and 2 GB pages

Long-running memory-intensive applications benefit most from using large pages. Short-lived processes with a small memory working set are not good candidates. The system programmer uses the LFAREA system parameter to define the amount of real storage that can be used for 2 GB pages and 1 MB fixed pages that are initially obtained via the IARV64 service. See IEASYSxx (system parameter list) in *z/OS MVS Initialization and Tuning Reference*.

Using assembler instructions in the 64-bit address space

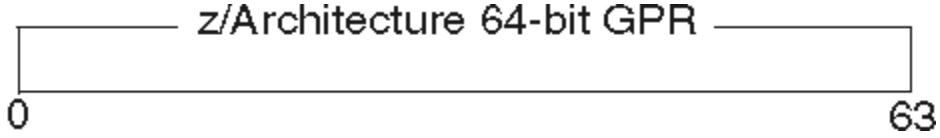
With z/Architecture, two facts are prominent: the address space is 16 exabytes in size, and the general purpose registers (GPRs) are 64 bits in length. You can ignore these facts and continue to use storage below the bar. If, however, you want to enhance old programs or design new ones to use the virtual storage above the bar, you will need to use the new Assembler instructions. This section introduces the concepts that provide context for your use of these instructions.

z/Architecture provides two new major capabilities that are related but are also somewhat independent:

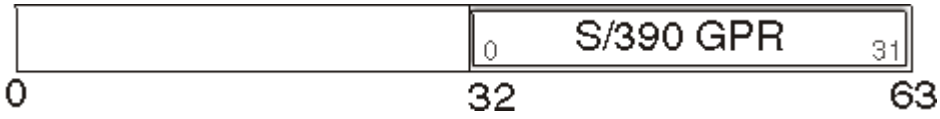
- 64-bit binary operations
- 64-bit addressing mode (AMODE).

64-bit binary operations

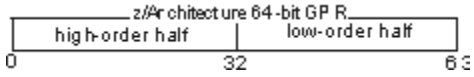
64-bit binary operations perform arithmetic and logical operations on 64-bit binary values. 64-bit AMODE allows access to storage operands that reside anywhere in the 16-exabyte address space. In support of both, z/Architecture extends the GPRs to 64 bits. There is a single set of 16 64-bit GPRs, and the bits in each are numbered from 0 to 63.



All S/390® instructions are carried forward into z/Architecture and continue to operate using the low-order half of the z/Architecture 64-bit GPRs. That is, an S/390 instruction that operates on bit positions 0 through 31 of a 32-bit GPR in S/390 operates instead on bit positions 32 through 63 of a 64-bit GPR in z/Architecture. You can think of the S/390 32-bit GPRs as being imbedded in the new 64-bit GPRs.



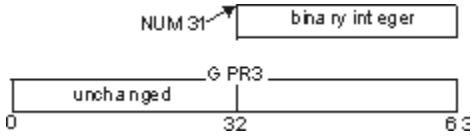
Throughout the discussion of GPRs, bits 0 through 31 of the 64-bit GPR are called the **high-order half**, and bits 32 through 63 are called the **low-order half**.



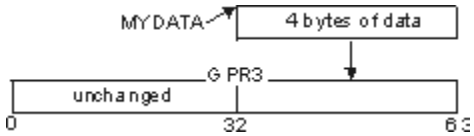
The purpose of this section is to help you use the 64-bit GPR and the 64-bit instructions as you want to save registers, perform arithmetic operations, access data. It is not a tutorial about how to use the new instruction set. *Principles of Operation* is the definitive reference book for these instructions. This section, however, describes some concepts that provide the foundation you need. After you understand these, you can go to *Principles of Operation* and read the introduction to z/Architecture that appears in the first chapter and then refer to the specific instructions you need to write your program.

How z/Architecture processes S/390 instructions

First of all, your existing programs work, unchanged, in z/Architecture mode. This section describes how z/Architecture processes S/390 instructions. The best way to describe this processing is through examples of common S/390 instructions. First, consider a simple Add instruction: A R3,NUM31. This instruction takes the value of a fullword binary integer at location NUM31 and adds it to the contents of the low-order half of GPR3, placing the sum in the low-order half of GPR3. The high-order half of GPR3 is unchanged.



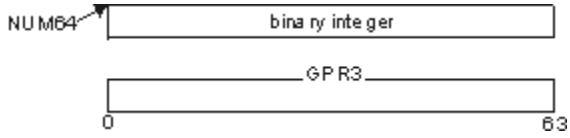
Second, consider the LOAD instruction: L R3,MYDATA. This instruction takes the 4 bytes of data at location MYDATA and puts them into the low order bits of GPR3.



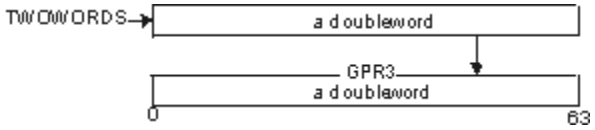
The high-order half is not changed by the ADD instruction or the LOAD instruction. The register forms of these instructions (AR and LR) work similarly, as do Add Logical instructions (AL and ALR).

z/Architecture instructions that use the 64-bit GPR

z/Architecture provides many new instructions that use two 64-bit binary integers to produce a 64-bit binary integer. These instructions include a "G" in the instruction mnemonic (AG and LG). Some of these instructions are similar to S/390 instructions. Consider the example of an Add G instruction: AG R3,NUM64. This instruction takes the value of a doubleword binary integer at location NUM64 and adds it to the contents of GPR3, placing the sum in GPR3:



The second example, LG R3,TWOWORDS, takes a doubleword at location TWOWORDS and puts it into GPR3.



Because 32-bit binary integers are prevalent in S/390, z/Architecture also provides instructions that use a 64-bit binary integer and a 32-bit binary integer. These instructions include a "GF" in the instruction mnemonic (AGF and LGF). Consider AGF. In AGF R3,MYDATA, assume that MYDATA holds a 32-bit positive binary integer, and GPR3 holds a 64-bit positive binary integer. (The numbers could have been negative.) The AGF instruction adds the contents of MYDATA to the contents of GPR3 and places the resulting signed binary integer in GPR3; the sign extension, in this case, is zeros.

The AGFR instruction adds the contents of the low-order half of a 64-bit GPR to bits 0 through 63 in another 64-bit GPR. Instructions that include "GF" are very useful as you move to 64-bit addressing.

64-bit addressing mode (AMODE)

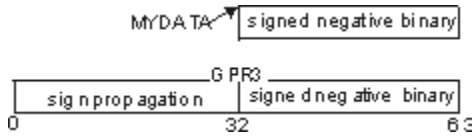
When generating addresses, the processor performs address arithmetic; it adds three components: the contents of the 64-bit GPR, the displacement (a 12-bit value), and (optionally) the contents of the 64-bit index register. Then, the processor checks the addressing mode and truncates the answer accordingly. For AMODE 24, the processor truncates bits 0 through 39; for AMODE 31, the processor truncates bits 0 through 32; for AMODE 64, no truncation (or truncation of 0 bits) occurs. In S/390 architecture, the processor added together the contents of a 32-bit GPR, the displacement, and (optionally) the contents of a 32-bit index register. It then checked to see if the addressing mode was 31 or 24 bits, and truncated accordingly. AMODE 24 caused truncation of 8 bits. AMODE 31 caused a truncation of bit 0.

The addressing mode also determines where the storage operands can reside. The storage operands for programs running in AMODE 64 can be anywhere in the 16-exabyte address space, while a program running in AMODE 24 can use only storage operands that reside in the first 16 megabytes of the 16-exabyte address space.

Non-modal instructions

An instruction that behaves the same, regardless of the AMODE of the program, is called a **non-modal** instruction. The only influence AMODE exerts on how a non-modal instruction performs is where the storage operand is located. Two excellent examples of non-modal instructions have already been described: the Load and the Add instructions. Non-modal z/Architecture instructions that are already described also include the LG instruction and the AGF instruction. For example, programs of any AMODE can issue AG R3,NUM64, described earlier, which adds the value of a doubleword binary integer at location NUM64 to the contents of GPR3, placing the sum in GPR3.

The LGF instruction is another example of a non-modal instruction. In LGF R3,MYDATA, assume MYDATA is a signed negative binary integer. This instruction places MYDATA into the low-order half of GPR3 and propagates the sign (1s) to the high-order half, as follows:



If the current AMODE is 64, MYDATA can reside anywhere in the address space; if the AMODE is 31, MYDATA must reside below 2 gigabytes; if the AMODE is 24, MYDATA must reside below 16 megabytes.

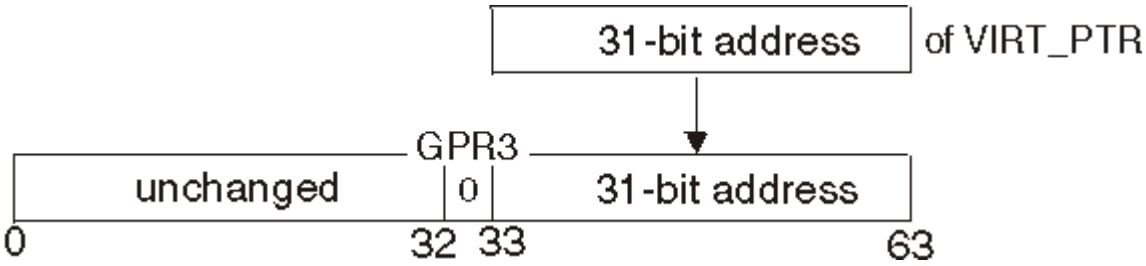
Other 64-bit instructions that are non-modal are the register form of AGF, which is AGFR, and the register form of LGF, which is LGFR. Others are LGR, AGR, ALGR, and ALG.

Modal instructions

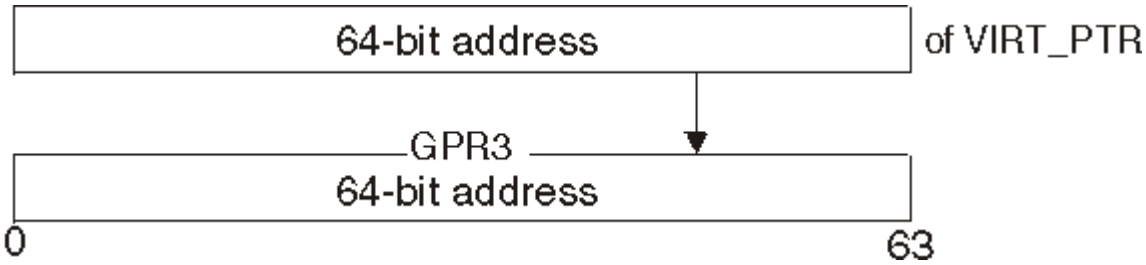
Modal instructions are instructions where the addressing mode is a factor in the output of the instruction. The AMODE determines the width of the output register operands. A good example of a modal instruction is Load Address (LA). If you specify LA R3,VIRT_PTR successively in the three AMODEs, what are the three results?

- *In AMODE 24*, the address of VIRT_PTR is a 24-bit address that is loaded into bits 40 through 63 of GPR3 (or bits 8 through 31 of the 32-bit register imbedded in the 64-bit GPR). The processor places zeros into bits 32 through 39, and leaves the first 31 bits unchanged, as follows:

- *In AMODE 31*, the address of VIRT_PTR is loaded into bits 33 through 63 of GPR3. The processor places zero into bit 32 and leaves the first 32 bits unchanged, as follows:



- *In AMODE 64*, the address of VIRT_PTR fill the entire 64-bit GPR3:



Other modal instructions are Move Long (MVCL), Branch and Link (BALR), and Branch and Save (BASR).

Setting and checking the addressing mode

z/Architecture provides three new Set Addressing Mode instructions that allow you to change addressing mode. The instructions are SAM24, which changes the current AMODE to 24, SAM31, which changes the current AMODE to 31, and SAM64, which changes the current AMODE to 64.

Starting with z/OS V1R3, there are other ways for a program to be in AMODE 64:

- If your program uses the assembler AMODE 64 statement, and is bound that way, then the load module is AMODE 64 and the system will give it control in AMODE 64.
- You could use the binder AMODE(64) statement to define that your load module is AMODE 64 and the system will give it control in AMODE 64.
- You could be a target PC routine and have set up the entry table entry to indicate that your routine is to be given control in AMODE 64.
- Your interface could be via BASSM and you could have set up an 8-byte target with the last bit on; callers can then load and then issue BASSM. Your routine would then be entered in AMODE 64.

The AMODE bits in the PSW tell the processor what AMODE is currently in effect. You can obtain the current addressing mode of a program by using the Test Addressing Mode (TAM) instruction. In response, TAM sets a condition code based on the setting in the PSW; 0 indicates AMODE 24, 1 indicates AMODE 31, and 3 indicates AMODE 64.

Linkage conventions

In z/OS R2, program entry is in AMODE 24 or AMODE 31; therefore linkage conventions you have used in S/390 apply, which means passing 4-byte parameter lists and a 72-byte save area.

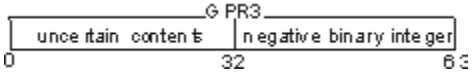
An older program changing from AMODE 31 to AMODE 64 to exploit z/Architecture instructions should expect to receive 31-bit addresses and the 72-byte save area from its callers. If you are running in AMODE 64 and want to use an address a caller has passed to you, the high-order half of the GPR will probably not be cleared to zeros. As soon as you receive this address, use the Load Logical G Thirty One Bits (LLGT or LLGTR) instruction to change this 31-bit address into a 64-bit address that you can use.

Pitfalls to avoid

As you begin to use the 64-bit instructions, consider the following:

1. Some instructions reference or change all 64 bits of a GPR regardless of the AMODE.
2. Some instructions reference or change only the low-order half of a GPR regardless of the AMODE.
3. Some instructions reference or change only the high-order half of a GPR regardless of the AMODE.
4. When you are using signed integers in arithmetic operations, you can't mix instructions that handle 64-bit integers with instructions that handle 31-bit integers. The interpretation of a 32-bit-signed number differs from the interpretation of a 64-bit-signed number. With the 32-bit-signed number, the sign is extended in the low half of the doubleword. With the 64-bit-signed number, the sign is extended to the left for the entire doubleword.

Consider the following example, where a 31-bit subtraction instruction has left a 31-bit negative integer in bits 32 through 63 of GPR3 and has left the high-order half unchanged.



Next, the instruction AG R3,MYDOUBLEWORD, mentioned earlier, adds the doubleword at the location MYDOUBLEWORD to the contents of the GPR3 and places the sum at GPR3. Because the high-order half of the GPR has uncertain contents, the result of the AG instruction is incorrect. To change the value in the GPR3 so that the AG instruction adds the correct integers, before you use the AG instruction, use the Load G Fullword Register (LGFR) instruction to propagate the sign to the high-order half of GPR3.

Using a memory object

To use the storage in a memory object, the program must be in AMODE 64. (See “Setting and checking the addressing mode” on page 70 for ways to get into AMODE 64.) While in AMODE 64, a program can issue the IARV64 macro to create and free memory objects and to manage the physical frames behind the storage. The parameter lists the program passes to IARV64 can reside above or below the bar.

To invoke macros other than those capable of being issued in AMODE 64, a program must be in AMODE 31 or AMODE 24. This restriction might mean that the program must first issue SAM31 to return to

AMODE 31. After a program issues a macro that is not capable of being issued in AMODE 64, it can return to AMODE 64 through SAM64. To learn whether a program is in AMODE 64, see "Setting and checking the addressing mode" on page 70.

Managing the data, such as serializing the use of a memory object, is no different from serializing the use of an area obtained through GETMAIN or STORAGE.

Although few macros can be issued in AMODE 64, other interfaces support storage above the bar. For example, the DUMP command with the ST0R=(*beg*,*end*[,*beg*,*end*]...) parameter specifies ranges of virtual storage to be dumped. Those ranges can be above the bar.

In summary, there are major differences between how you manage storage below the bar and how you manage storage above the bar. Table 7 on page 72 can help you understand the differences and some of the similarities. The first column identifies a task or concept, the second column applies to storage below the bar, and the third column applies to storage above the bar.

Table 7. Comparing tasks and concepts for memory objects: Below the bar and above the bar

Task or concept	**Below the bar**	**Above the bar**
Obtaining storage	GETMAIN, STORAGE, CPOOL macros and callable cell pool services. On GETMAIN and STORAGE, you can request to have a return code indicate whether the storage is cleared to zeros.	IARV64 GETSTOR request creates private memory objects; storage is cleared to zeros. IARV64 GETSHARED request creates shared memory objects; storage is cleared to zeros. IARV64 GETCOMMON request creates 64-bit common memory objects; storage is cleared to zeros.
Creating a 64-bit common memory object	Not applicable.	The request can be conditional or unconditional. The request specifies the size of the memory object in megabytes.
Increments of storage allocation	In 8-byte increments, on a double-word boundary.	In megabyte increments, on a megabyte boundary.
Requirements for requestor	GETMAIN with BRANCH=NO cannot be issued by an SRB or a program in AR mode. GETMAIN with BRANCH=YES can be issued by an SRB or a program in AR mode. STORAGE can be issued by an SRB or by a program in AR mode. CPOOL can be issued by an SRB but not by a program in AR mode. Callable cell pool services can be issued in either mode and by an SRB.	IARV64 GETSTOR can be issued in SRB mode and in AR mode. The invoker of the IARV64 service must be enabled and hold local lock when issuing the GETSTOR request. There is no restriction on FRRs. IARV64 GETCOMMON can be issued in task mode or in AR mode. The invoker of the service can be enabled or disabled.
Freeing storage	FREEMAIN, STORAGE, CPOOL macros, and callable cell pool services. Any 8-byte increment of the originally-obtained storage can be freed. An entire subpool can be freed with a single request. At task termination, storage owned by task is freed; some storage (common, for example) does not have an owner.	Use the IARV64 DETACH request. Memory objects can only be freed in their entirety; no partial detaches are allowed. All memory objects obtained with a specified user-defined token can be freed with a single request. At task termination, private storage owned by task is freed; all private storage has an owner and that owner is a task. For shared memory objects, see "Freeing a shared memory object" on page 83.
Page fixing virtual storage and making those pages available to be paged out	PGSER FIX request and PGSER FREE request for any storage.	IARV64 PAGEFIX and IARV64 PAGEUNFIX requests.
Notifying the system of an anticipated use of storage	PGSER LOAD request PGSER OUT request.	IARV64 PAGEIN and IARV64 PAGEOUT requests.
Making a range of storage read-only or modifiable	PGSER PROTECT request and PGSER UNPROTECT request.	Use the IARV64 CHANGEACCESS request for shared memory objects.

Table 7. Comparing tasks and concepts for memory objects: Below the bar and above the bar (continued)		
Task or concept	**Below the bar**	**Above the bar**
Discard data in physical pages and optionally clear the pages to zeros.	PGSER RELEASE request clears the storage to zeros. Note that PGRLSE, PGSER RELEASE, PGSER FREE with RELEASE=Y, and PGFREE RELEASE=Y may ignore some or all of the pages in the input range, and will not notify the caller if this was done. Any pages in the input range that match any of the following conditions will be skipped, and processing continues with the next page in the range: • Storage is not allocated or all pages in a segment have not yet been referenced. • Page is in PSA, SQA or LSQA. • Page is V=R. Effectively, it's fixed. • Page is in BLDL, (E)PLPA, or (E)MLPA. • Page has a page fix in progress, or a nonzero FIX count. • Pages with COMMIT in progress or with DISASSOCIATE in progress.	IARV64 DISCARDDATA request. CLEAR=YES must be specified to guarantee the storage is cleared to zeros on the next usage.
Obtaining information about use of storage areas	VSMLIST service.	IARV64 LIST request.
Storage key and fetch protection attributes	Apply to the entire allocated area.	Apply to the entire allocated area.
What the area consists of	System programs and data, user programs and data.	User data only.
Performing I/O	VSAM, BSAM, BPAM, QSAM, VTAM, Media Manager, and EXCP and EXCPVR services.	EXCP and EXCPVR requests and Media Manager.
Accessing storage	To access data in the 2-gigabyte address space, a program must run in AMODE 31 or AMODE 64. S/390 and z/Architecture instructions can be used.	To access data in the 16-exabyte address space, a program must run in AMODE 64. To load an address of a location above the bar into a GPR, a program must use a z/Architecture instruction.

IARV64 macro services

The IARV64 macro provides services to manage the 64-bit virtual storage for your programs. Table 8 on page 73 introduces these services and the rules for what programs can do with the memory objects your programs create and use. The first column identifies the particular IARV64 request, the second column describes what a program can do if it is in problem state or has PSW key 8 - F, and the third column describes what a program can do if it is in supervisor state or has PSW key 0 - 7.

Table 8. IARV64 service requests and rules for programs working with memory objects		
IARV64 request	**A program in problem state, key 8 - F**	**A program in supervisor state or key 0 - 7**
GETSTOR — Create a private memory object	Can get a private memory object in the primary address space, only when the program's home and primary address space is the same. The storage key of the memory object will be the same as the PSW key of the caller. Can assign ownership of the memory object to the TCB of the job step task or the mother task (the task of the program that issued the ATTACHX).	Can get a private memory object in the primary or home address space, as specified by ALETVALUE. Can assign ownership of the private memory object to a TCB, specified by the TTOKEN, in the address space indicated by ALETVALUE. Can define the storage key of the private memory object. Can specify whether the private memory object can be freed by an unauthorized program and whether it can be pagefixed and unpagefixed.

Table 8. IARV64 service requests and rules for programs working with memory objects (continued)

IARV64 request	A program in problem state, key 8 - F	A program in supervisor state or key 0 - 7
GETSHARED — Create a shared memory object	Cannot use this service.	Can create shared memory objects in the primary or home address space.
GETCOMMON — Create a common memory object	Cannot use this service.	Can use this service in the primary or home address space.
DETACH — Free one or more memory objects	Can free a memory object it owns.	Can free a memory object it owns. Can free a memory object it does not own if the memory object is in the primary or home address space of the program issuing in IARV64.
PAGEFIX — Fix physical pages If you specify a list of page ranges, PAGEFIX can fix the physical pages that back more than one nonshared memory object.	Cannot fix pages.	Can fix pages in one or more nonshared memory objects in the primary or home address space.
UNPAGEFIX — Undo a PAGEFIX operation	Cannot unfix pages.	Can unfix pages in one or more nonshared memory objects in the primary or home address space.
PAGEOUT — Alert the system that physical pages will not be used so that the system can optimize the use of the physical pages	Can use only if the memory object is in the primary address space.	Can use for pages that back memory objects in the primary or home address space.
PAGEIN — Alert the system that pages will be needed soon	Can use only if the memory object is in the primary address space.	Can use for pages that back memory objects in the primary or home address space.
DISCARDATA — Discard data in physical pages and optionally clear the pages to zeros. Also, you can optionally free the real frames for these pages. If you specify a list of page ranges, DISCARDDATA can discard data in more than one memory object.	Can use if all of the following conditions are true: • The PSW key of caller is the same as the storage key of the memory object. • The memory object is in the primary address space.	Can use for memory objects in the primary or home address space. Can use if the PSW key does not match the storage key of the memory object.
CHANGEGUARD — See "Creating guard areas and changing their sizes" on page 88	Can use this service only if it owns the memory object or if an ancestor task or the job step task owns the memory object.	Can use this service in the primary or home address space.
LIST — List the memory objects	Cannot list memory objects.	Can list memory objects in the primary or home address space.
SHAREMEMOBJ — Request that the specified address space be given access to one or more shared memory objects	Cannot use this service.	Can use this service in the primary or home address space.
CHANGEACCESS — Request that a view type for segments within the specified shared memory objects be changed	Cannot use this service.	Can use this service in the primary or home address space.
CHANGEATTRIBUTE — Change an attribute of storage within one or more memory objects.	Can request to change an attribute of storage within one or more memory objects that have a storage key that matches the PSW key.	Can request to change an attribute of storage within one or more memory objects.

Table 9 on page 75 lists the various IARV64 requests and whether they are valid for private, shared, and common memory objects.

Table 9. IARV64 services valid for private, shared, and common memory objects			
IARV64 request	**Memory object was obtained with this IARV64 request:**		
	GETSTOR	**GETSHARED**	**GETCOMMON**
PAGEFIX	Yes, with restrictions. See the IARV64 service in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.	No	Yes, with restrictions. See the IARV64 service in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.
PAGEUNFIX	Yes, when CONTROL=AUTH is specified on the GETSTOR request	No	Yes
PAGEOUT	Yes	Yes	Yes
PAGEIN	Yes	Yes	Yes
DISCARDDATA	Yes	Yes	Yes
CHANGEGUARD	Yes	No	No
PROTECT	Yes	No	Yes
UNPROTECT	Yes	No	Yes
LIST	Yes	Yes	Yes
COUNTPAGES	Yes	Yes	Yes
DETACH	Yes	Yes	Yes
SHAREMEMOBJ	No	Yes	No
CHANGEACCESS	No	Yes	No
CHANGEATTRIBUTE	Yes	Yes	Yes

In summary, an unauthorized program can:

- Create memory objects in its own address space and relate them to each other
- Issue a CHANGEGUARD for memory objects it owns, or for memory objects that an ancestor task or its job-step task owns
- Page out and page in the physical pages that back the memory objects it owns
- Discard data in the physical pages that back the memory objects it owns
- Detach the memory objects it owns.

This document discusses how to use the IARV64 services, but does not describe environmental or programming requirements, register usage, or syntax rules. For that information, see the description of the IARV64 macro in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Protecting storage above the bar

To limit access to the memory object, the creating program can use the FPROT and KEY parameters on IARV64. KEY assigns the storage key for the memory object, and FPROT specifies whether the storage in the memory object is fetch-protected. Storage protection and fetch protection attributes apply for the entire memory object. A program can only reference storage in a fetch-protected memory object that runs with the same PSW key as the storage key of the memory object or PSW key 0.

Tagging 64-bit memory objects for data privacy

To control the distribution of sensitive data in 64-bit memory objects, the creating program can use the SENSITIVE parameter on the IARV64 service. SENSITIVE=YES indicates that the memory object contains sensitive data. Tagged sensitive data in dumps can be secured and redacted when post processed by Data

Privacy for Diagnostics (DPfD). For more information about DPfD, see *z/OS MVS Diagnosis: Tools and Service Aids*.

- Consider tagging memory objects as SENSITIVE=YES when they contain data of a personal or confidential nature that can cause harm to the individual or business if not safeguarded, such as regulated data as defined by General Data Protection Regulation (GDPR), Health Insurance Portability and Accountability Act (HIPAA), or other legal requirements.
- Consider tagging memory objects as SENSITIVE=NO when they do not contain data of a personal or confidential nature.
- Consider tagging memory objects as SENSITIVE=UNKNOWN, which is the default, when you are unsure of the sensitive nature of the data.
- IARV64 REQUEST=CHANGEATTRIBUTE can be used to specify different sensitive states for subsections of the memory object, but there will be higher system memory overhead than for a memory object with a uniform SENSITIVE setting.

Creating private memory objects

The GETSTOR request of the IARV64 service creates private memory objects.

GETSTOR request

To create a memory object, use the IARV64 GETSTOR request. However, when you create a memory object, request a size large enough to meet long-term needs; the system abends a program that unconditionally tries to obtain more storage above the bar than the MEMLIMIT allows. IBM recommends that you specify COND=YES on the request to avoid the abend. In this case, if the request exceeds the MEMLIMIT, the system rejects the request, but the program continues to run. The IARV64 service returns to the caller with a nonzero return code. The recovery routine would be similar to one that would respond to unsuccessful STORAGE macro conditional requests for storage.

The SEGMENTS parameter specifies the size, in megabytes, of the memory object you are creating. The system returns the address of the memory object in the ORIGIN parameter.

Parameters for GETSTOR include:

- FPROT=YES gives it fetch protection.
- SENSITIVE is an optional parameter that specifies whether the memory object contains sensitive data (for instance, personal or confidential data), as described in Tagging 64-bit memory objects for data privacy.
- KEY=*key* specifies its storage key (authorized programs only).
- TTOKEN=*ttoken* indicates what task is to own the memory object.
- ALETVALUE=*alet* identifies the address space in which the memory object is to reside, either the home or primary address space (authorized users only).
- USERTKN=*user token* is an 8-byte token that relates two or more memory objects to each other. Later, the program can request a list of memory objects that have that same token and can delete them as a group.
- SVCDUMPRGN=YES specifies that the storage in the memory object is to be included when an SVC dump is requested through SDUMPX SDATA=(RGN). SVCDUMPRGN=NO specifies that the virtual storage in the memory object is not to be included when an SVC dump is requested through SDUMPX SDATA=(RGN). There are other ways to include this storage in the dump, such as the SDUMPX SUMLIST64 or SDUMPX LIST64 requests.
- LOCALSYSAREA=YES specifies that the storage is not copied to the child address space when the POSIX FORK system call is invoked. In general, this option is appropriate for system-related control blocks like that a 31-bit application would obtain out of high private. Only authorized invokers can specify LOCALSYSAREA=YES. When this option is specified the size of the obtained object is not subject to the MEMLIMIT threshold.

- LOCALSYSAREA=NO specifies that the storage is copied to the child address space when the POSIX FORK system call is invoked. In general, this option is appropriate for application-related storage that is analogous to storage that a 31-bit application would obtain out of region.
- MEMLIMIT=YES specifies that the request is subject to MEMLIMIT threshold.
- MEMLIMIT=NO specifies that the request is not subject to MEMLIMIT threshold. If you need to specify this option, consider whether LOCALSYSAREA=YES should also be specified.
- DUMP=LIKELSQA indicates that the memory object should be included in an SVC dump when LSQA is included. Strongly consider this option if LOCALSYSAREA=YES is specified.
- DETACHFIXED=YES indicates that it is acceptable to detach the memory object with fixed pages. If the application intends to use the storage like fixed LSQA (for example, fix once and leave it fixed for the life of the object), then this option might be appropriate to avoid having to unfix the storage before detaching it.
- SADMP=YES specifies that the memory object is to be captured in a stand-alone dump.
- SADMP=NO Specifies that the memory object is not captured in a stand-alone dump unless explicitly requested by the stand-alone dump program. Consider this option if LOCALSYSAREA=NO is specified.
- EXECUTABLE=YES indicates that code is able to be executed from the obtained storage.
- EXECUTABLE=NO indicates that code will not be able to be executed from the obtained storage on a system that has implemented Instruction_Execution_Protection.
- CONTROL=AUTH prevents a memory object from being freed by an unauthorized program (authorized users only). Additionally, CONTROL=AUTH is required if you plan to fix and unfix pages.
- INORIGIN=*inorigin* specifies the address of the desired storage to be obtained.

When a program creates a memory object, it can specify, through the GUARDSIZE and GUARDHIGH and GUARDLOW parameters, that the memory object is to consist of two different areas. One area is called a guard area; this storage is not accessible; the other area is called the usable area. A later request can change the guard area into a usable area. The section "Creating guard areas and changing their sizes" on page 88 can help you understand the important purposes for this kind of memory object.

Before issuing IARV64, issue SYSSTATE ARCHLVL=2 so that the macro generates the correct parameter addresses.

For a complete description of the IARV64 GETSTOR request, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Example of creating a private memory object

The following example creates a memory object one megabyte in size. It specifies a constant with value of one as a user token.

```
IARV64 REQUEST=GETSTOR,
       SEGMENTS=ONE_SEG,
       USERTKN=USER_TOKEN,
       ORIGIN=VIRT64_ADDR,
       COND=YES
ONE_SEG     DC  ADL8(1)
USER_TOKEN  DC  ADL8(1)
VIRT64_ADDR DS  AD
```

Freeing a private memory object

When your program no longer needs the memory object, it uses IARV64 DETACH to free (delete) the memory object. You can free memory objects that are related to each other through the user token defined on the IARV64 GETSTOR request. Additionally, all programs can use the following parameters:

- MATCH=SINGLE,MEMOBJSTART frees a specific memory object, as identified by its origin address.
- MATCH=USERTKN, USERTKN frees a related set of memory objects by providing the user token specified when the memory objects were created.

- COND=YES makes the request conditional, but only when you also pass a user token. IBM recommends you use COND to avoid having the program abend because it asked to free a memory object that doesn't exist.

Authorized programs can use additional parameters:

- ALETVALUE frees all memory objects in the primary address space or the home address space.
- OWNER=YES,TTOKEN frees only memory objects that are owned by a specified task.
- OWNER=NO (without TTOKEN) frees memory objects regardless of which task owns them.

Three conditions to avoid when you try to free a memory object are:

- Freeing a memory object that does not exist.

  If you try to free a memory object that doesn't exist, the system abends your program.
- Freeing a memory object that has a range of addresses PAGEFIXED.

  If you try to free a memory object that has a range of addresses pagefixed, the system will abend and address space termination might follow. See the paragraph about task information in this section.
- Freeing a memory object that has I/O in progress.

If you specify the COND=YES parameter, you must also specify a user token. In the recovery routine that gets control at an abend, you can try one of the following:

- Unfix any fixed pages. If you can unfix the pages, you can try again to free the memory object.
- Ignore the abend and leave the memory object in an unusable state.

As part of normal task termination, RSM frees the memory objects owned by the terminating task; if RSM determines that there are fixed pages in the memory object, the system issues a CALLRTM TYPE=MEMTERM request that results in address space termination. To avoid this MEMTERM, your recovery routine should try to terminate any active I/O into the memory object that your program created and free any pages that your program fixed.

Example of freeing a private memory object

The program frees all memory objects that have the user token specified in "USER_TOKEN":

```
IARV64 REQUEST=DETACH,
       MATCH=USERTOKEN,
       USERTKN=USER_TOKEN
USER_TOKEN DC  ADL8(1)
```

An example of creating, using, and freeing a private memory object

The following program creates a 1-megabyte memory object and writes the character string "Hi Mom" into each 4k page of the memory object. The program then frees the memory object.

```
         TITLE 'TEST CASE DUNAJOB'
         ACONTROL FLAG(NOALIGN)
DUNAJOB  CSECT
DUNAJOB  AMODE 31
DUNAJOB  RMODE 31
         SYSSTATE ARCHLVL=2
* Begin entry linkage
         BAKR  14,0
         CNOP  0,4
         BRAS  12,@PDATA
         DC    A(@DATA)
@PDATA   LLGF  12,0(12)
         USING @DATA,12
         LHI   0,DYNAREAL
         STORAGE  OBTAIN,LENGTH=(0),SP=0,CALLRKY=YES
         LLGTR 13,1
         USING @DYNAREA,13
         MVC   4(4,13),=C'F6SA'
* End entry linkage
*
         SAM64                     Change to amode64
```

```
         IARV64 REQUEST=GETSTOR,                                       +
               SEGMENTS=ONE_SEG,                                       +
               USERTKN=USER_TOKEN,                                     +
               ORIGIN=VIRT64_ADDR
         LG    4,VIRT64_ADDR       Get address of memory obj
         LHI   2,256               Set loop counter
LOOP     DS    0H
         MVC   0(10,4),=C'HI_MOM!' Store HI MOM!
         AHI   4,4096
         BRCT  2,LOOP
* Get rid of all memory objects created with this
* user token
         IARV64 REQUEST=DETACH,                                        +
               MATCH=USERTOKEN,                                        +
               USERTKN=USER_TOKEN,                                     +
               COND=YES
*
* Begin exit linkage
         LHI   0,DYNAREAL
         LR    1,13
         STORAGE RELEASE,LENGTH=(0),ADDR=(1),SP=0,CALLRKY=YES
         PR
* End exit linkage
@DATA    DS    0D
ONE_SEG  DC    FD'1'
USER_TOKEN DC  FD'1'
         LTORG
@DYNAREA DSECT
SAVEAREA DS    36F
VIRT64_ADDR DS AD
DYNAREAL EQU   *-@DYNAREA
         END   DUNAJOB
```

Creating shared memory objects

This section explains how to create shared memory objects:

- The “GETSHARED request” on page 79 of the IARV64 service creates shared memory objects that can be shared across address spaces.
- The “SHAREMEMOBJ request” on page 81 allows an address space to access the shared memory object.
- The “CHANGEACCESS request” on page 82 manages the type of access that is allowed to the shared memory object.

For a complete description of the IARV64 macro, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

GETSHARED request

To create a shared memory object, use the IARV64 GETSHARED service. While shared memory storage is like common storage in that it is shared across address spaces, it differs because there is not automatic addressability or access to it.

The shared memory object is allocated when you use the GETSHARED service. To access data in a shared memory object you need to use the IARV64 GETSHARED request to create the shared memory object, and the IARV64 “SHAREMEMOBJ request” on page 81 to specify an interest in the shared memory object.

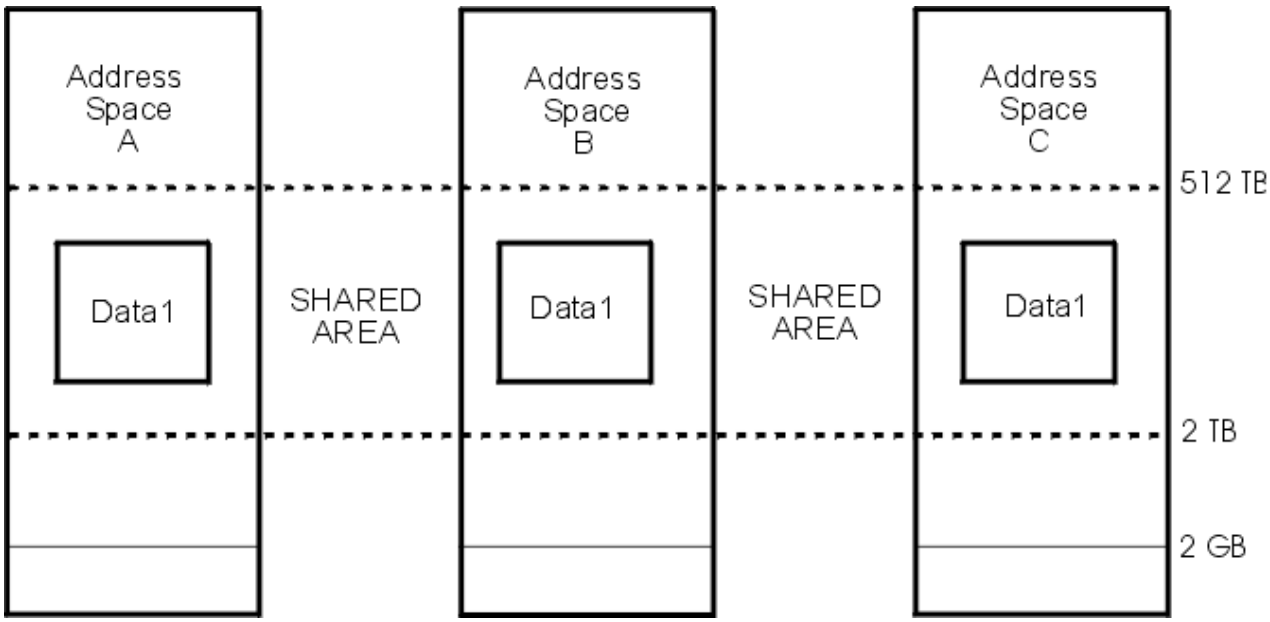


Figure 20. z/OS R5 Address Space with the default shared area between 2-terabytes and 512-terabytes

Notice that when you create a shared memory object, you specify some of the same attributes as when you create a non-shared memory object through the GETSTOR service.

Parameters for the IARV64 GETSHARED service include:

- SEGMENTS=*segments* specifies the size, in megabytes, of the shared memory object you are creating.
- FPROT=YES gives the memory object fetch protection. The default is FPROT=NO.
- SENSITIVE is an optional parameter that specifies whether the memory object contains sensitive data (for instance, personal or confidential data), as described in Tagging 64-bit memory objects for data privacy.
- KEY=*key* specifies its storage key. The key must be in bits 0-3 of the specified byte.
- CHANGEACCESS=*scope* identifies whether subsequent CHANGEACCESS requests change the access for only the address space specified (LOCAL), or all the address spaces sharing the memory object (GLOBAL). The default is CHANGEACCESS=LOCAL.
- USERTKN=*usertoken* is a required 8-byte token that relates two or more memory objects to each other. Associate the user token with the memory object, so later you can free several shared memory objects at one time.
- ORIGIN= *origin* is the name or address that will contain the lowest address in the memory object.

For an example of this request, see "Example of creating and using a shared memory object – GETSHARED" on page 82.

Relationship between the shared memory object and its owner

When your program creates shared memory objects you need to understand ownership issues in order to prevent illegal operations that will be identified by an ABEND from z/OS. A program creates a shared memory object, but it does not own the shared memory object. A shared memory object is always owned by the system. A program gains access to a shared memory object by creating an interest in the shared memory object. Shared interest is owned by the CMRO (cross memory resource owner) TCB of an address space. Once a program has gained access to a shared memory object, any program running in that address space has access to the shared memory object.

If the unit of work is an SRB, the program must assign ownership to a TCB. Because of this assignment of ownership, the owner of the memory object and the creator of the memory object might not always be the same.

The memory object is available to programs with the correct PSW key and ALET value. The memory object can be accessed by programs running under the owning TCB and other programs running in the same address space. A program can use a memory object in its primary address space if its PSW key matches the storage key of the memory object. An authorized program can use a memory object in another address space if it has the ALET for that address space on its access list and if its PSW key matches the storage key of the memory object.

When an address space terminates, or when a batch job completes execution under an initiator, the system removes any shared interest the address space had in the shared memory objects. If the terminating address space was the only address space with an interest in the shared memory object and the system affinity has been removed from the shared memory object, the system deletes the shared memory object. The memory object is no longer available for use.

When a TCB terminates, the system deletes the memory objects that the TCB owns. The system swaps a memory object in and out as it swaps in and out the address space that dispatched the owning TCB.

A memory object can remain active even after the creating TCB terminates if a program assigns ownership of the memory object to a TCB that will outlive the creating TCB. In this case, termination of the creating TCB does not affect the memory object. To illustrate the importance of assigning ownership to the appropriate TCB, consider the following example:

- PGMA, a program running in its home address space AS1, issues the PC instruction to call PGMB which runs in AS2.
- While in AS2, PGMB creates a memory object and assigns ownership to TCB2 in AS2. PGMB also ALESERV ADDs AS2 to PGMA's dispatchable unit access list (DUAL), so that PGMA can continue to reference the memory object after PGMB PR's back to PGMA in AS1.
- After PGMB issues the PR to return to AS1, PGMA continues to use the memory object created by PGMB. Sometime later, TCB2 terminates and the system deletes the memory object. The next time PGMA references the memory object, the system issues an abend.

When assigning ownership of a memory object to a TCB, make sure the owning TCB will exist for the life of its address space and the memory object will exist for the life of the TCB. Such a TCB would be the TCB that owns the cross memory resources of the address space; the address of the cross memory owning TCB of AS2 is stored in the ASCBXTCB field of the ASCB of AS1.

SHAREMEMOBJ request

To get access to the shared memory object, a program uses the SHAREMEMOBJ service. An address space can issue more than one SHAREMEMOBJ request for the same memory object. To separate each of the requests for the same memory object you need to specify a different user token.

Parameters for the IARV64 SHAREMEMOBJ service include:

- USERTKN=*user token* uniquely identifies the user token to be associated with the memory object. For a single shared memory object, the given user token is allowed to be duplicated in distinct address spaces, but not allowed to be duplicated within a single address space.
- RANGLIST=*ranglistptr* specifies an address pointing to a list of memory objects that the program wants to access.
- NUMRANGE=*numrange* specifies the number of entries in the supplied range list pointed to by RANGLIST. You can specify up to 16 memory objects.
- ALETVALUE=*aletvalue* indicates the ALET of the address space that will access the memory object.
- SVCDUMPRGN=*YES/NO* specifies whether or not the shared memory object is included in an SVC dump. The default is SVCDUMPRGN=YES.

For an example of this request, see “Example of accessing a shared memory object – SHAREMEMOBJ” on page 83.

CHANGEACCESS request

To request that the type of access to the specified virtual storage be changed, use the CHANGEACCESS request. The three types of access are:

- READONLY
- SHAREDWRITE
- HIDDEN

The scope of the change is determined by your choice of LOCAL or GLOBAL on the CHANGEACCESS parameter of the GETSHARED service. When CHANGEACCESS=LOCAL is specified or defaulted on the CHANGEACCESS parameter of the GETSHARED service, only the address space specified by the ALET parameter on the CHANGEACCESS service is affected. If the shared memory object is not addressable by this address space because the IARV64 SHAREMEMOBJ request was not issued, the request will fail.

When the CHANGEACCESS=GLOBAL is specified on the CHANGEACCESS parameter of the GETSHARED service, all address spaces currently sharing the memory object are affected, so all address spaces will get the same view. Subsequent IARV64 SHAREMEMOBJ requests for this memory object will also be affected until the next CHANGEACCESS invocation. Memory objects with CHANGEACCESS=GLOBAL support CHANGEACCESS requests without prior SHAREMEMOBJ requests.

CHANGEACCESS requests for memory objects specified as CHANGEACCESS=LOCAL require that the target space have interest in the shared memory object, which means a SHAREMEMOBJ for the target space must be done before the CHANGEACCESS request. Because CHANGEACCESS changes the view of the specified address space you must have access to memory object before you can CHANGEACCESS.

Parameters for the IARV64 CHANGEACCESS service include:

- VIEW specifies the type of access you want to have to the virtual storage. The three types of access are READONLY, SHAREDWRITE, or HIDDEN.
- RANGLIST specifies the virtual address which can be anywhere in the shared memory object. The following rules apply:
  - The starting address must be on a segment boundary.
  - The starting address must be within a memory object returned by a GETSHARED request.
  - A single range must be contained within a single memory object.
- ALETVALUE=*aletvalue* indicates the ALET of the address space that will access the memory object.
- NUMRANGE specifies the number of entries in the supplied range list.

For an example of this request, see "Example of changing the access of a shared memory object – CHANGEACCESS" on page 83.

Examples using IARV64 requests for shared memory objects

Example of creating and using a shared memory object – GETSHARED

The following example creates a shared memory object one megabyte in size. It specifies a constant with value of one as a user token.

```
IARV64 REQUEST=GETSHARED,
       SEGMENTS=ONE_SEG,
       USERTKN=USERTKNA,
       ORIGIN=VIRT64_ADDR,
       COND=YES,
       FPROT=NO,
       KEY=MYKEY,
       CHANGEACCESS=LOCAL
ONE_SEG    DC FD'1'
USERTKNA    DC 0D'0'
            DC F'15' High Half must be non-zero
              DC F'1'  UserToken of 1
VIRT64_ADDR DS  D
```

Note: If you want the memory object to have key 9, the declaration for MYKEY is as follows:

```
MYKEY  DC X'90'
```

Example of accessing a shared memory object – SHAREMEMOBJ

The following example allows access to a shared memory object:

```
IARV64 REQUEST=SHAREMEMOBJ,
       USERTKN=USERTKNS,
       RANGLIST=RLISTPTR,
       NUMRANGE=1,
       ALETVALUE=0,
       COND=YES,
       SVCDUMPRGN=YES
USERTKNS      DC 0D'0'
              DC F'15'    High Half Must Be Non-Zero
              DC F'2'     User Token of 2
RLISTPTR    DS   AD       Pointer to the IARV64 Parmlist


```

Example of changing the access of a shared memory object – CHANGEACCESS

The following example changes the type of access to virtual storage to HIDDEN for the primary address space:

```
IARV64 REQUEST=CHANGEACCESS,
       VIEW=HIDDEN,
       RANGLIST=RLISTPTR,
       NUMRANGE=1,
       ALETVALUE=0
RLISTPTR    DS   AD       Pointer to the IARV64 Parmlis
```

Freeing a shared memory object

To free a shared memory object, use the IARV64 DETACH request. All address spaces have to remove interest from the memory object by issuing a DETACH AFFINITY=LOCAL request. The system interest is removed from the memory object by issuing a DETACH AFFINITY=SYSTEM request.

AFFINITY=LOCAL

When you specify the AFFINITY=LOCAL parameter, the system uses the specified or defaulted ALET to determine if the address space has access to the memory object identified by the USERTKN parameter. Then one of the following can happen:

- If no other sharers of the memory object remain (either the current address space or other address spaces) and a detach with AFFINITY=SYSTEM has been done for the memory object, the memory object is freed and no longer available for use.
- If other sharers of the memory object remain, or detach with AFFINITY=SYSTEM has not been done for the memory object, the memory object is not freed, but the interest identified by the USERTKN parameter value is removed. If this is the last sharer of the object for the address space specified by the ALET parameter value, the address space will no longer have access to the memory object.

The following example frees the address space interest for the memory object specified by the user token:

```
IARV64 REQUEST=DETACH,
       AFFINITY=LOCAL,
       ALETVALUE=0,
       COND=YES,
       MATCH=SINGLE,
       MEMOBJSTART=VIRT64_ADDR,
       USERTKN=USERTOKEN
VIRT64_ADDR   DS AD
USERTOKEN     DC XL8'E2C8C1D9E3D6D2D5'     Value is SHARTOKN
```

AFFINTY=SYSTEM

When you specify AFFINTY=SYSTEM, the system interest for the memory object identified by the USERTKN parameter is removed. The shared memory object will be freed when there is no remaining interest in the object.

The following example frees the system interest:

```
IARV64 REQUEST=DETACH,
       AFFINITY=SYSTEM,
       COND=YES,
       MATCH=SINGLE,
       MEMOBJSTART=VIRT64_ADDR,
       USERTKN=USERTOKEN
VIRT64_ADDR   DS AD
USERTOKEN     DC XL8'E2C8C1D9E3D6D2D5'     Value is SHARTOKN
```

Proper serialization of shared memory objects

It is important to serialize access to shared memory objects so that the shared memory objects your program creates and uses do not cause situations where z/OS finds reason to abnormally end the work unit. Shared memory objects can be serialized with ENQs, Latches, or other cross-address space serialization techniques.

The following example shows the behavior of memory objects when strict serialization is not maintained.

Example: Tasks A, B, and C are sharing storage through a GETSHARED request obtained by task A. Task B and C are continually sharing the storage via SHAREMEMOBJ and DETACH AFFINITY=LOCAL. They continue sharing until task A chooses to DETACH AFFINITY=SYSTEM to the shared storage while both of the following conditions apply:

- Task B still holds an interest in the storage and
- Task C no longer has a local interest in the shared storage (issued DETACH AFFINITY=LOCAL).

If work unit C tries to share the memory object again (through SHAREMEMOBJ), the system will issue an abend code DC2 with reason code xx0040xx because it was not serialized against the DETACH AFFINITY=SYSTEM issued by work unit A.

Creating common memory objects

Memory objects created in 64-bit common virtual storage are visible at the same virtual address in every address space in the system and are accessible by every address space in the system. They have a single protection key (only keys 0 - 7 are allowed) and fetch protection attribute.

A 64-bit common memory object can be created via the IARV64 REQUEST=GETCOMMON request. When no longer needed, 64-bit common memory objects must be explicitly freed via the IARV64 REQUEST=DETACH request with AFFINITY=SYSTEM.

The following chart summarizes the attributes of 64-bit common memory objects, compared to 64-bit shared memory objects and below-the-bar common storage.

Like common storage below 2 GB:	**Unlike common storage below 2 GB:**
• Memory objects are visible at the same address in every address space. • Every address space has access to the memory object once it is created. • Storage tracker capabilities are provided for diagnostic purposes. • Memory objects can be pageable, fixed, or DREF. • Memory objects must be explicitly freed.	• Memory objects cannot be implicitly fixed at allocation. • 64-bit common storage has no subpools.

Like 64-bit shared memory objects:	Unlike 64-bit shared memory objects:
• Memory objects are allocated in 1 MB multiples on a 1 MB boundary. • Memory objects can be grouped as a set of related objects by specifying a memory object token. • Memory objects must be explicitly freed.	• Virtual storage is addressable by all address spaces at allocation time. • There is no need to explicitly request access to the virtual storage. • Memory objects can be explicitly fixed. • Memory objects can be referenced while disabled (DREF).

GETCOMMON request

To create a 64-bit common memory object, use the IARV64 GETCOMMON request. Your program must be running in supervisor state and key 0 - 7, and the memory object to be obtained can only be assigned system keys 0 - 7.

The following parameters are of particular interest on a GETCOMMON request:

- MOTKNSOURCE is an optional parameter that indicates the source of the memory object token that is to be associated with this memory object. A memory object token can be supplied on a later DETACH request in order to free all of the memory objects that are associated with that token.
  - MOTKNSOURCE=SYSTEM means that the system will provide the memory object token in the output area identified by the OUTMOTKN parameter. This token can be used on subsequent GETCOMMON requests as a user-supplied token in order to associate other memory objects with this token.
  - MOTKNSOURCE=USER, which is the default, means that the user is providing the memory object token, specified by the MOTKN parameter. This must be a token that was returned by the system in the OUTMOTKN parameter of a previous GETCOMMON request. If you do not specify the MOTKN parameter, no user token is supplied to associate this memory object with others.
- TYPE is an optional parameter that specifies the type of storage being requested. The TYPE parameter is honored when PAGEFRAMESIZE=4K is specified, or when PAGEFRAMESIZE=MAX is specified and the memory object is backed with 4 KB page frames. The TYPE parameter is ignored when PAGEFRAMESIZE=1MEG is specified, or when PAGEFRAMESIZE=MAX is specified and the memory object is backed with 1 MB page frames.
  - TYPE=PAGEABLE, which is the default, indicates that pages backing this memory object will be pageable. Pages will be backed upon first reference and can be paged out to auxiliary storage. Virtual address ranges within the memory object can be explicitly fixed after allocation by using the IARV64 PAGEFIX request.
  - TYPE=DREF indicates that the memory object will be referenced while running disabled. The DREF attribute applies to the entire memory object. Pages will be backed upon first reference and will remain in real storage; they will never be paged out to auxiliary storage.
- OWNERCOM is an optional parameter that specifies the entity to which the system will assign ownership of the common memory object. The system uses this ownership information to track the use of 64-bit common storage for diagnostic purposes.
  - OWNERCOM=HOME, which is the default, indicates that the home address space will be assigned as the owner of the common memory object.
  - OWNERCOM=PRIMARY indicates that the primary address space will be assigned as the owner of the common memory object.
  - OWNERCOM=SYSTEM indicates that the system will be assigned as the owner of the common memory object; the memory object is not associated with an address space.
  - OWNERCOM=BYASID indicates that the address space specified by the OWNERASID parameter will be assigned as the of the common memory object.
- DUMP is an optional parameter that specifies whether the 64-bit common memory object is to be included in an SVC dump when CSA or SQA is specified on SDATA.

  - DUMP=LIKECSA, which is the default when TYPE=PAGEABLE is specified on the GETCOMMON request, indicates that the common memory object is to be included in an SVC dump when CSA is specified on SDATA.
  - DUMP=LIKESQA, which is the default when TYPE=DREF is specified on the GETCOMMON request, indicates that the common memory object is to be included in an SVC dump when SQA is specified on SDATA.
  - DUMP=NO indicates that the common memory object is not to be included in an SVC dump when either CSA or SQA is specified on SDATA.
  - DUMP=BYOPTIONVALUE indicates that the common memory object is to be dumped according to the option specified by the OPTIONVALUE parameter.
- DETACHFIXED is an optional parameter that specifies whether the common memory object can be detached when it contains fixed pages at the time of the DETACH request.
  - DETACHFIXED=NO, which is the default, indicates that the memory object will not be detached if it contains any fixed pages at the time of the DETACH request.
  - DETACHFIXED=YES indicates that the memory object will be detached even if some or all of its pages are fixed.
- SADMP=YES specifies that the memory object is to be captured in a stand-alone dump.
- SADMP=NO Specifies that the memory object is not captured in a stand-alone dump unless explicitly requested by the stand-alone dump program.
- EXECUTABLE=YES indicates that code is able to be executed from the obtained storage.
- EXECUTABLE=NO indicates that code will not be able to be executed from the obtained storage on a system that has implemented Instruction_Execution_Protection.
- SENSITIVE is an optional parameter that specifies whether the memory object contains sensitive data (for instance, personal or confidential data), as described in Tagging 64-bit memory objects for data privacy.

For a complete description of the IARV64 GETCOMMON request, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Freeing a common memory object

To free a 64-bit common memory object, use the IARV64 DETACH request. Your program must be running in supervisor state and key 0 - 7.

The following parameters are of particular interest on a DETACH request for common memory objects:

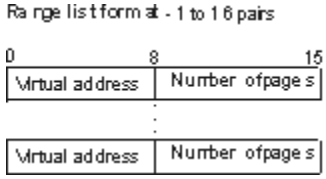
- MATCH is an optional parameter that specifies which memory objects are to be freed.
  - MATCH=SINGLE, which is the default, indicates that a single memory object, specified by the MEMOBJSTART parameter, is to be freed. MEMOBJSTART contains the address of the first byte in the memory object.
  - MATCH=MOTOKEN indicates that memory objects associated with a memory object token are to be freed. The memory object token is specified by the MOTKN or USERTKN parameter. Memory objects that are not associated with a memory object token are not affected.
  - MATCH=USERTOKEN is a synonym for MATCH=MOTOKEN.
- MOTKN is an optional parameter that specifies the memory object token that uniquely identifies the memory objects. The MOTKN parameter is used along with the MOTKNCREATOR parameter, and is mutually exclusive with the USERTKN parameter.
- MOTKNCREATOR is an optional parameter that indicates who created the memory object token specified by the MOTKN parameter.
  - MOTKNCREATOR=USER, which is the default, indicates that the memory object token specified by MOTKN is user-created.
  - MOTKNCREATOR=SYSTEM indicates that the memory object token specified by MOTKN is system-created.

- USERTKN is an optional parameter that specifies the user-created memory object token that uniquely identifies the memory objects. The USERTKN parameter is mutually exclusive with the MOTKN parameter, and is equivalent to MOTKN with MOTKNCREATOR=USER.
- AFFINITY is an optional parameter that specifies whether local or system affinity for the memory object is to be affected.
  - AFFINITY=LOCAL, which is the default, indicates that local affinity to the memory object is to be affected. This does not apply to common memory objects.
  - AFFINITY=SYSTEM indicates that system affinity to the memory object is to be affected, and is only applicable to shared memory objects and common memory objects. You must specify AFFINITY=SYSTEM on a DETACH request for common memory objects.
- V64COMMON is an optional keyword that specifies whether this DETACH request is for 64-bit common memory objects.
  - V64COMMON=NO, which is the default, indicates that this DETACH request is not for 64-bit common memory objects.
  - V64COMMON=YES indicates that this DETACH request is for 64-bit common memory objects. You must specify V64COMMON=YES on a DETACH request for common memory objects.

For a complete description of the IARV64 DETACH request, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Fixing the pages of a memory object

Authorized programs can use IARV64 PAGEFIX to fix specified 4K pages in a single memory object. Page fixing prevents the system from stealing those pages. The ALETVALUE parameter tells the system where the memory object resides: the primary or the home address space. The LONG parameter tells the system the pagefix is expected to be of long duration (in seconds). On the RANGLIST parameter, the program provides a list of the page ranges that are to be fixed. The format of the list is:



IARV64 PAGEUNFIX requests that the system unpagefix the pages that back memory objects. A page remains fixed until the number of unpagefix operations for that page equals the number of pagefix operations. As with the pagefix request, the ALETVALUE parameter to specify that the memory object is in the primary or home address space and the program provides a list of page ranges.

Example of fixing pages of a memory object

Using the memory object created earlier, the following example in an AMODE 31 program, fixes 5 pages of the memory object, then unfixes them:

```
         SYSSTATE ARCHLVL=2
         .
         .
         .
         XC    R_LIST(100),R_LIST      Clear the range list
         LG    12,VIRT64_ADDR          Get starting address to pagefix
         STG   12,R_START              Save it in range list
         LGHI  4,5                     Load number of pages to fix
         STG   4,R_PAGES               Save it in range list
         SLR   12,12                   Generate primary-space alet
         ST    12,R_ALET               Save it in range list
         LA    4,R_LIST                Get address of rangelist
         LLGTR 4,4                     Make it a 64-bit pointer
         STG   4,RLISTPTR              Save it
* Now pagefix the 5 pages
         IARV64 REQUEST=PAGEFIX,                                       +
               RANGLIST=RLISTPTR,                                      +
               LONG=NO
```

```
* Using the same rangelist, unfix the pages
         LA    12,R_LIST              Get address of range list
         LLGTR 12,12                  Make it a 64-bit pointer
         STG   12,RLISTPTR            Save it
         IARV64 REQUEST=PAGEUNFIX,                                    +
               RANGLIST=RLISTPTR
*
*   Declares for example
R_LIST   DS    CL100
         ORG   R_LIST
R_START  DS    ADL8
R_PAGES  DS    ADL8
R_ALET   DS    AL4
RLISTPTR DS    AD
VIRT64_ADDR DS AD
```

Discarding data in a memory object

Your program can use the IARV64 DISCARDDATA request to tell the system that your program no longer needs the data in certain pages and that the system can free them. Optionally, you can use the CLEAR parameter to clear the area to zeros. Also you can optionally use the KEEPREAL parameter to specify whether the real frames backing the pages to be discarded are to be freed or not. The RANGLIST parameter provides a list of page ranges, as shown earlier.

Authorized programs can use the ALETVALUE parameter to specify that the memory objects are in the primary address space or the home address space.

Releasing the physical resources that back pages of memory objects

A program uses the IARV64 PAGEOUT request to tell the system that the data in certain pages will not be used for some time (as measured in seconds) and that the pages are candidates for paging out of real storage. A pageout does not affect pages that are fixed in real storage. On the RANGLIST parameter, the program provides a list of page ranges. Authorized programs can use the ALETVALUE parameter to designate memory objects in the address space identified by the ALET.

A program uses the IARV64 PAGEIN request to tell the system that it will soon reference the data in certain pages and that the system should page them into real storage if the pages are not already backed by real storage. Authorized programs can use the ALETVALUE parameter to target pages of memory objects in the address space identified by the ALET.

Creating guard areas and changing their sizes

A program can create a memory object that consists of two areas:

- An area it can use immediately, called the *usable area*
- A second area, called a *guard area*

The system does not allow programs to use storage in the guard area. Additional guard areas within the memory object can also be created.

To create a memory object with a guard area, use the IARV64 GETSTOR or GETCOMMON request with either the SEGMENTS or UNITS parameter to specify the size of the memory object, and either the GUARDSIZE or GUARDSIZE64 parameter to specify the size of the initial guard area. Use GUARDLOC=LOW or GUARDLOC=HIGH to specify whether the initial guard area is to be at the low end or the high end of the memory object.

Note: A request to create a guard area when the range contains fixed pages results in an abend.

Use a guard area to reserve the area for future use. For example, a program can manage the parceling out of pages of the memory object. Another reason for using a guard area is so that the program requesting the memory object can protect itself from accidentally referencing storage beyond the end of the memory object, and possibly overlaying data in another adjacent memory object. For that, the program would use GUARDLOC=HIGH. If the program wanted to protect itself from another program that might be using an adjacent memory at a lower address, it would likely use GUARDLOC=LOW.

Use COND=YES when invoking the IARV64 REQUEST=CHANGEGUARD request, conditionally requesting the change, to avoid an abend if the request exceeds the MEMLIMIT established by the installation. If it cannot grant a conditional request, the system rejects the request, but the program continues to run.

The following illustration shows a memory object, eight segments in size. GUARDLOC=HIGH creates the guard area at the highest addresses of the memory object. The memory object has seven segments of usable storage and one segment on reserve for later use.

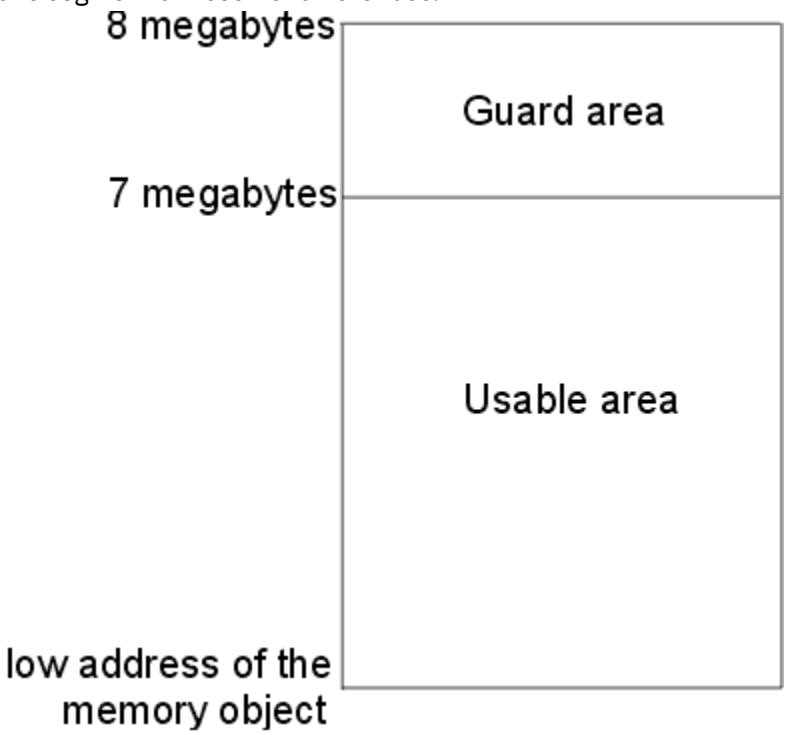


Figure 21. A memory object eight megabytes in size

Use the IARV64 CHANGEGUARD request to increase or decrease the amount of usable space in a memory object by adjusting the size of the initial guard area or creating and removing additional guard areas. Use the MEMOBJSTART keyword to increase or decrease the initial guard area. For private area memory objects, use the CONVERTSTART keyword to create, increase, or decrease guard areas at specific addresses. If any storage of the request is already in the requested state, usable or guarded, CHANGEGUARD processes the request and returns a code of four as a warning. Common area memory objects can only have a low or high guard area.

The following illustration shows an additional guard area of two segments being created in the already defined private area memory object. This guard area is being created at a specific address, three segments within the memory object. Now the memory object has five segments of usable storage and three segments of unusable (guarded) storage.

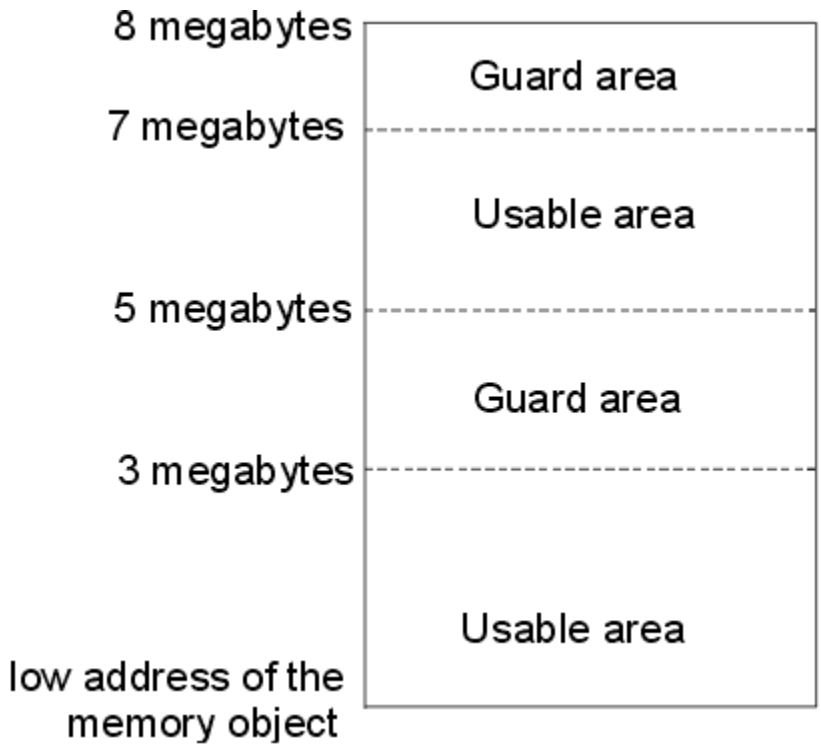


Figure 22. A memory object with an additional guard area

Your program cannot reference an address in the guard areas; if it does, the program receives a program exception (0C4 abend). To avoid abend, code a recovery routine to get control upon receiving the program exception; the recovery routine can retry and then increase the usable part of the memory object thus decreasing the guard areas.

The guard areas do not count towards the MEMLIMIT set by the installation; the usable areas do count toward the MEMLIMIT.

Examples of creating a memory object with a guard area

The following example creates a 3-megabyte memory object with a 2-megabyte guard area. The guard area is at the high end of the memory object:

```
IARV64 REQUEST=GETSTOR,
        SEGMENTS=NUM_SEG,
        USERTKN=USER_TOKEN,
        GUARDSIZE=GUARDPAGES,
        GUARDLOC=HIGH,
        CONTROL=AUTH,
        ORIGIN=VIRT64_ADDR
```

The following example decreases the size of the guard area by the specified amount.

```
IARV64 REQUEST=CHANGEGUARD,
        CONVERT=FROMGUARD,
        MEMOBJSTART=VIRT64_ADDR,
        CONVERTSIZE=SEGMENT_SIZE,
        ALETVALUE=0
```

The following example creates a guard area at a specific address within the memory object:

```
IARV64 REQUEST=CHANGEGUARD,
       CONVERT=TOGUARD,
       CONVERTSTART=GUARD_START,
       CONVERTSIZE=SEGMENT_SIZE,
       ALETVALUE=0
```

Listing information about the use of virtual storage above the bar

Authorized programs can use the IARV64 LIST request to obtain information about memory objects in the caller's address space. The system returns the information in a work area you provide. The V64LISTPTR parameter defines the first address of this work area; the V64LISTLENGTH identifies the length of the area. The parameter list macro is mapped by IAXV64WA.

The system returns the following information about usable areas (not guard areas) of memory objects:

- Beginning address
- Ending address
- Storage key
- Shared or private indicator
- Multiple guard areas indicator

To request a list of shared memory objects defined for the system via GETSHARED specify V64SHARED=*YES*.

Changing the attributes of storage within a memory object

A program uses the IARV64 CHANGEATTRIBUTE request to alter storage attributes of a subsection of one or more memory objects. On the RANGLIST parameter, the program provides a list of page ranges. Authorized programs can use the ALETVALUE parameter to designate memory objects in the address space identified by the ALET.

Dumping 64-bit common memory objects

To include 64-bit common memory objects in an SVC dump:

- Specify a list of 64-bit address ranges to include in the dump using the LIST64 parameter on the SDUMPX macro.

  Any 64-bit common memory object can be dumped this way. This is also the only way to dump 64-bit common memory objects that specified DUMP=NO on the IARV64 GETCOMMON request.
- Specify SDATA=(CSA,SQA).
  - 64-bit common memory objects that were allocated with a dump attribute of DUMP=LIKECSA on the IARV64 GETCOMMON request will be included in an SVC dump when CSA is specified on the SDATA parameter.
  - 64-bit common memory objects that were allocated with a dump attribute of DUMP=LIKESQA on the IARV64 GETCOMMON request will be included in an SVC dump when SQA is specified on the SDATA parameter.

The 64-bit common storage will be dumped after common storage below 2 GB and before private storage.

Chapter 5. Using access registers

The term "extended addressability" refers to the ability of a program to use virtual storage that is outside the address space the program is dispatched in. Chapter 3, “Synchronous cross memory communication,” on page 19 describes how a caller uses the PC instruction to call a program in another address space and run there under the caller's TCB. It describes the two cross memory instructions (MVCS and MVCP) that move data from primary to secondary and from secondary to primary.

Access registers provide you with a different function from cross memory. You cannot use them to branch into another address space. Through access registers, however, you can use assembler instructions to manipulate data in other address spaces and in data spaces. You do not use access registers to reference addresses in hiperspaces.

In addition to this section, other sources of information can help you understand how to use access registers:

- Chapter 6, “Creating and using data spaces,” on page 127, contains examples of using access registers to manipulate data in data spaces.
- *Principles of Operation* contains descriptions of how to use the instructions that manipulate the contents of access registers.

Also, the following books contain the syntax and parameter descriptions for the macros that are mentioned in this section:

- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

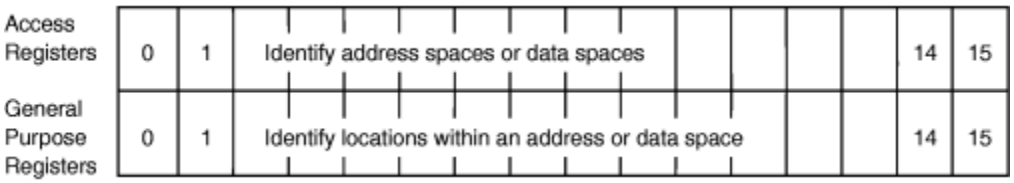
Using access registers for data reference

Through access registers, your program, whether it is supervisor state or problem state, can use assembler instructions to perform basic data manipulation, such as:

- Comparing data in one address space with data in another
- Moving data into and out of a data space, and within a data space
- Accessing data in an address space that is not the primary address space
- Moving data from one address space to another
- Performing arithmetic operations with values that are located in different address spaces or data spaces.

The functions of cross memory and access registers are different and complementary. In a multiple address space environment, you might use them both.

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 general purpose registers (GPRs).



Why would a program use ARs? Generally, instructions and data reside in a single address space — the primary address space (PASN). However, you might want your program to have more virtual storage than a single address space offers, or you might want to separate data from instructions for:

- Storage isolation and protection
- Data security
- Data sharing among multiple users

For these reasons and others, your program can have data in address spaces other than the primary or in data spaces. The instructions still reside in the primary address space, but the data can reside in another address space or in a data space.

To access data in other address spaces, your program uses ARs and executes in the address space control mode called access register mode (**AR mode**).

What is address space control (ASC) mode? The ASC mode determines where the system looks for the data that the address in the GPR indicates. The two ASC modes that are generally available for your programs are primary and AR mode. The PSW determines the ASC mode. Both problem state and supervisor state programs can use both modes, and a program can switch between the two modes.

- **In primary mode**, the data your program can access resides in the program's primary address space. (An exception to this statement is that a program in primary mode can use the cross memory instructions, MVCP and MVCS, to manipulate data in the secondary address space.) When it resolves the addresses in data-referencing instructions, the system does not use the contents of the ARs.
- **In AR mode**, the data your program can access resides in the address/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/data space. Specifically, the AR contains a value, called an ALET, that identifies the address space or data space that contains the data, and the GPR contains a base address that points to the data within the address/data space. (In this document, the term **address/data space** refers to "address space or data space".)

The following chart summarizes where the system looks for the instructions and the data when the program is in primary mode and AR mode.

ASC Mode	Location of Instructions	Location of Data
Primary mode	Primary address space	Primary address space
AR mode	Primary address space	Address/data space identified by an AR

In this document, the AR and GPR pair that is used to resolve an address is called **AR/GPR**. Figure 23 on page 94 illustrates AR/GPR 4.

Figure 23. Example of an AR/GPR

Do not confuse cross memory mode with ASC mode. A program can be in AR mode with the primary, secondary, and home address spaces all the same. Likewise, a program can be in AR mode with the primary, secondary, and home address spaces all different. Chapter 3, “Synchronous cross memory communication,” on page 19 contains information about cross memory mode.

Do not confuse addressing mode (AMODE) with ASC mode. A program can be in AR mode and also be in either 31-bit or 24-bit addressing mode. However, programs in 24-bit addressing mode are restricted in their use of data spaces; for example, a program in 24-bit addressing mode cannot create a data space, nor can the program access data above 16-megabytes in that space.

How does your program switch ASC mode? Use the SAC instruction to change ASC mode:

- SAC 512 sets the ASC mode to AR mode
- SAC 0 sets the ASC mode to primary mode

What does the AR contain? The contents of an AR designate an address/data space. The AR contains a token that specifies an entry in a table called an **access list**. Each entry in the access list identifies an address/data space that programs can reference. The token that indexes into the access list is called an **access list entry token (ALET)**. When an ALET is in an AR and the program is in AR mode, the ALET identifies the access list entry that points to an address/data space. The corresponding GPR contains the address of the data within the address/data space. **IBM recommends** that you use ARs only for ALETs and not for other kinds of data.

The following figure shows an ALET in the AR and the access list entry that points to the address/data space. It also shows a GPR that points to the data within the address/data space.

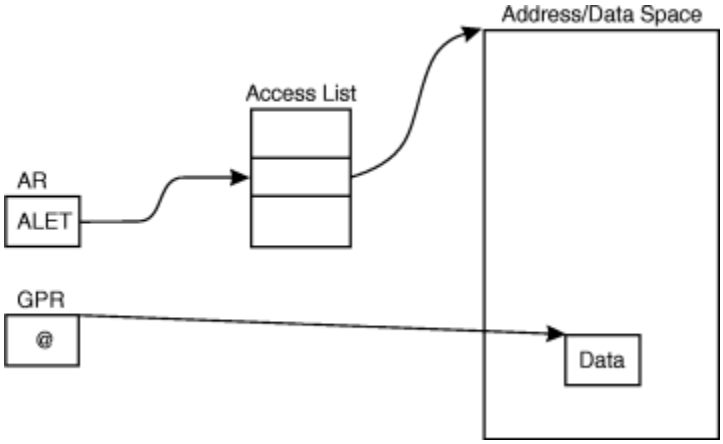


Figure 24. Using an ALET to identify an address/data space

By placing an entry on an access list and obtaining an ALET for that entry, a program builds the connection between the program and the target address/data space. (In describing the subject of authorization, the terms "target address space" and "target data space" are used to mean an address space or data space in which a program is trying to reference data.) The process of building this connection is called **establishing addressability** to an address/data space.

For programs in AR mode, when the GPR is used as a base register, the corresponding AR **must** contain an ALET. Conversely, when the GPR is not used as a base register, the corresponding AR is ignored. For example, the system ignores an AR when the associated GPR is used as an index register.

A comparison of data reference in primary and AR mode

The best way to show how address resolution in primary mode compares with address resolution in AR mode is through an example. Figure 25 on page 96 and Figure 26 on page 96 show two ways an MVC instruction works to move data at location B to location A.

In Figure 25 on page 96, the move instruction, MVC, is in code that is running in primary mode. The MVC instruction uses GPRs 1 and 2. GPR 1 is used as a base register to locate the destination of the MVC instruction. GPR 2 is used as a base register to locate some data to be moved.

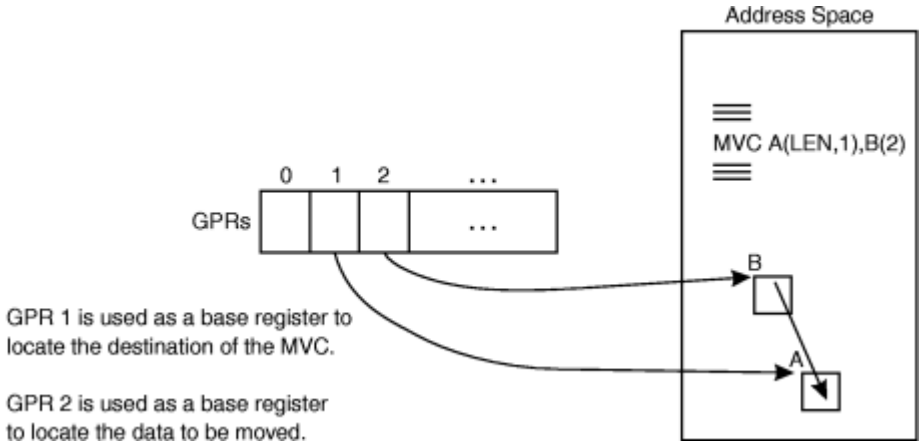


Figure 25. The MVC instruction in primary mode

In Figure 26 on page 96, the MVC instruction, in code that is in AR mode, moves the data at location B in Space Y to location A in Space X. 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, the MVC instruction is in code that is running in AR mode. The MVC instruction moves 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.

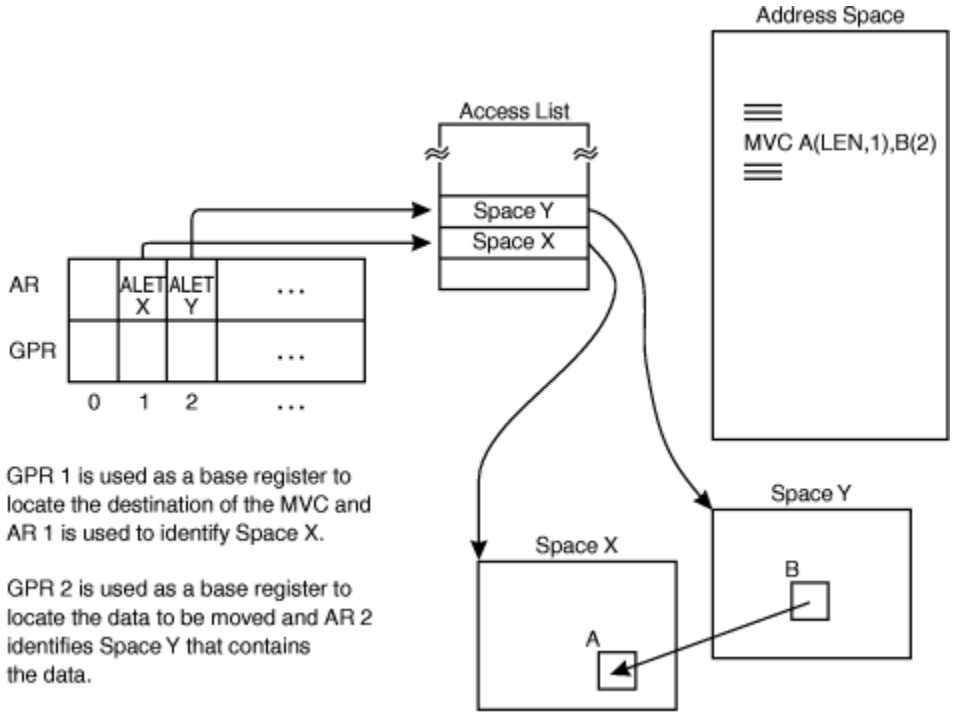


Figure 26. The MVC Instruction in AR Mode

Addresses that are qualified by an ALET are called **ALET-qualified** addresses.

Coding instructions in AR mode

As you write your AR mode programs, use the advice in this section.

- Always remember that for an instruction that uses a GPR as a base register, the system uses the contents of the associated AR to identify the address/data space that contains the data that the 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.
- You cannot use the following instructions when your program is in AR mode:
  - Move to primary — MVCP
  - Move to secondary — MVCS
  - Program transfer — PT
  - Basic program call — basic PC

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.

Table 10 on page 97 shows four examples of how a misplaced comma can change how the assembler resolves addresses on the load instruction.

Table 10. 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 Table 10 on page 97:

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 Table 10 on page 97:

In primary mode, the last two 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, the program can use assembler instructions to save, restore, and modify the contents of the 16 ARs. Both problem state and supervisor state programs can use these instructions.

The set of instructions that manipulate ARs includes:

- CPYA — Copy the contents of one AR into another AR.
- EAR — Copy the contents of an AR into a GPR.
- LAE — Load a specified ALET/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 location.
- STAM — Store contents of one or more ARs at a specified location.

For their syntax and help with how to use them, see the *Principles of Operation* documentation for your CPC.

Example of 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 you can use the LAM instruction to load an ALET into an AR.

The following instruction loads an ALET (located at DSALET) into AR 2:

```
        LAM   2,2,DSALET          LOAD ALET OF DATA SPACE INTO AR2
        .
DSALET  DS    F                   DATA SPACE ALET
```

Access lists

When the system first dispatches any work unit (such as a TCB, SRB, or IRB), it gives that work unit an access list (a DU-AL) that is empty. When the system creates an address space, it gives that address space an access list (PASN-AL) that contains only entries for existing data spaces that are known as common area (SCOPE=COMMON) data spaces. Programs add entries to the DU-AL and the PASN-AL. The entries represent the address/data spaces that the programs want to access.

Before your program can use ARs to reference data in an address/data space, it must establish a connection to the address/data space. The connection between the program that the work unit represents and the address/data spaces is through an access list. The process of establishing this connection is called establishing addressability.

Although you cannot use ARs to access data in hiperspaces, you can establish a connection between a program and a hiperspace through ALETs and access lists. If you are using hiperspaces see “Accessing hiperspaces” on page 169. The information in this section applies to data/address spaces.

Establishing addressability to an address/data space means your program must:

- Have authority to access data in the address/data space
- Have an access list entry that points to the address/data space
- Have the ALET that indexes to the entry

Before you can set up the access list entries and obtain ALETs, you need to know about:

- The two types of access lists, and the differences between them
- The two types of entries in access lists, and the differences between them
- The ALETs that are available to every program
- The ALESERV macro, which manages entries in access lists and gives information about ALETs and STOKENs.

The term **STOKEN** (for "space token") identifies an address space, a data space, subspace, or a hiperspace. It is similar to an address space identifier (ASID or ASN), with two important differences: the system does not reuse the STOKEN value within an IPL, and data spaces, subspaces, and hiperspaces do not have ASIDs. The STOKEN is an eight-byte variable that the system generates when you create an address space, data space, subspace, or hiperspace. (Note that the system never generates a STOKEN value of zero.)

Types of access lists

The access list can be one of two types:

- A primary address space access list (PASN-AL) — the access list that is associated with an address space
- A dispatchable unit access list (DU-AL) — the access list that is associated with a work unit (a TCB or SRB).

A program uses the DU-AL associated with its work unit and the PASN-AL associated with its primary address space.

The difference between a PASN-AL and a DU-AL is significant. If your program is a part of a subsystem that provides services for many users and has its own address space, it might reference address/data spaces through its PASN-AL. A program can create a data space, add an entry for the data space to the PASN-AL, and obtain the ALET that indexes the entry. By passing the ALET to other programs in the address space, the program can share the data space with other programs running in the address space.

If your program is not part of a subsystem, it will probably place entries for address/data spaces in its DU-AL.

Each work unit has one DU-AL; programs that the work unit represents can use it. That DU-AL cannot be shared with another work unit. A program can, however, use the ALCOPY parameter on the ATTACH(X) macro at the time of the attach, to pass a copy of its DU-AL to the attached task. "Attaching a subtask and sharing data spaces with it" on page 147 describes a program attaching a subtask and passing a copy of its DU-AL. This action allows two programs, the issuer of the ATTACH macro and programs running under the attached task, to have access to the address/data spaces that were represented by the entries on the DU-AL at the time of the attach.

Each address space has one PASN-AL. All programs running in the primary address space can use the PASN-AL for that address space. They cannot use the PASN-AL of any other address space.

The following lists summarize the characteristics of DU-ALs and PASN-ALs.

- The DU-AL has the following characteristics:
  - Each work unit has its own unique DU-AL.
  - All programs that the work unit represents can add and delete entries on the work unit's DU-AL.
  - A program cannot pass its task's DU-AL to a program running under another task. Tasks can never share a DU-AL. The one exception is that a program can pass a copy of its DU-AL to an attached task.

    When the DU-AL contains address space, data space, or hiperspace entries, the new subtask starts with an identical copy of the attaching task's DU-AL. The two DU-ALs do not necessarily stay identical. After the attach, the attaching task and the subtask are free to add and delete entries on their own DU-ALs.

    If the attaching task deletes the data space and the DU-AL entry for that data space, the subtask will still have an entry in its own DU-AL for that data space, but no program will be able to access this data space from the subtask.

    When the DU-AL contains subspace entries, the new subtask does not start with an identical copy of the attaching task's DU-AL, because the system does not copy the subspace entries to the subtask's DU-AL.
  - A program can pass its work unit's DU-AL to an SRB routine that the program schedules by using one of the following:

    - MODE=FULLXM parameter on the SCHEDULE macro
    - ENV=FULLXM parameter on the IEAMSCHD macro

    Similarly, a program can pass its work unit's DU-AL to a task that the program attaches by using the following:

    - ALCOPY=YES parameter on the ATTACH(X) macro

    The system dispatches the SRB with an identical copy of the scheduling/attaching work unit's DU-AL, minus any subspace entries, which are not copied. The new work unit (once it is dispatched) may add and delete entries on its DU-AL but must not delete the entries present on the DU-AL when it was initially dispatched.

    If the new work unit deletes initially-present access list entries, message IEF356I may be issued during job termination:

    - IEF356I ADDRESS SPACE UNAVAILABLE DUE TO CROSS MEMORY BIND

  - A DU-AL can have up to 509 entries.
  - A program can add more than one entry to its DU-AL for the same data space.
- The PASN-AL has the following characteristics:
  - Every address space has its own PASN-AL. The system initializes the PASN-AL to contain entries for existing SCOPE=COMMON data spaces.
  - Supervisor state programs and programs in PSW key 0 - 7 running with this address space as the primary address space can add and delete entries on the PASN-AL.
  - Problem state programs with PSW key 8 - F can add an entry to the PASN-AL for a SCOPE=SINGLE data space.
  - All programs running with this address space as the primary address space can access address/data spaces through the PASN-AL.
  - The PASN-AL is useful for cross memory service providers.
  - A PASN-AL can have up to 510 entries, some of which are reserved for SCOPE=COMMON data spaces.
  - When the job step terminates, the PASN-AL is purged.

Adding and deleting DU-AL and PASN-AL entries for address spaces might require that the program have special authorization. For more information on this authorization, see “EAX-authority to an address space” on page 115.

Because access lists belong to work units, you must remember the relationship between the program and the work unit that represents the program. For simplicity, this section describes access lists as if they belong to programs. For example, "your program's DU-AL" means "the DU-AL that belongs to the TCB that represents your program".

A Comparison of a PASN-AL and a DU-AL

Figure 27 on page 101 shows PGM1 that runs in AS1. The figure shows AS1's PASN-AL and PGM1's DU-AL. PGM1 shares the PASN-AL with other programs that execute in AS1. It does not share its DU-AL with any other programs. The PASN-AL contains entries to address/data spaces that program(s) placed there. PGM1 (either problem or supervisor state) has an entry for Space X to its DU-AL and an ALET for Space X. PGM1 received an ALET for Space Y from a program in supervisor state. Assuming PGM1 has authority to Space X and Space Y, it has addressability to Space X through its DU-AL and Space Y through its PASN-AL; it can access data in both Space X and Space Y. Therefore, with one MVC instruction, PGM1 can move data from a location in Space X to a location in Space Y.

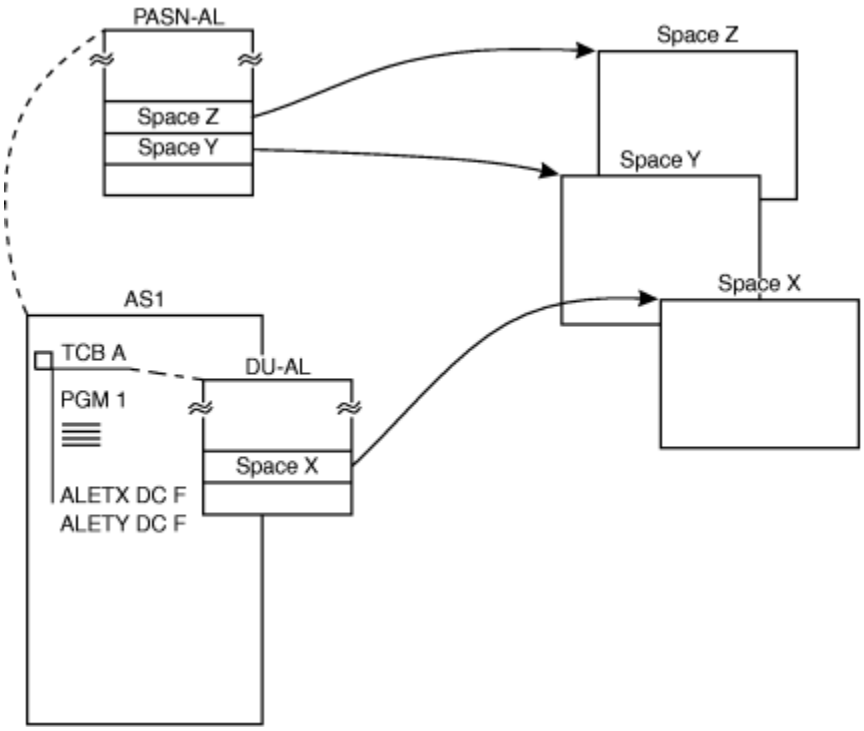


Figure 27. Comparison of addressability through a PASN-AL and a DU-AL

What happens to a program's PASN-AL and DU-AL at an address space switching operation? When a program issues a PC instruction that causes the primary address space to change, the system does not dispatch a new TCB or SRB. The PC routine that gets control runs under the same TCB or SRB as the program that issued the PC. Therefore, the PC routine keeps the DU-AL that is associated with the original program. However, the PC routine has the PASN-AL of the new primary address space. This is a different PASN-AL than the one used by the original program (except for the entries for SCOPE=COMMON data spaces).

For example, consider PGM1 in Figure 27 on page 101. It has addressability to Space X through TCB A's DU-AL and Space Y through its PASN-AL. Figure 28 on page 102 shows PGM1 issuing a PC to PGM2 in another address space. The figure shows how addressability through PASN-ALs and DU-ALs changes over a space-switching PC instruction. After the PC instruction, the PC routine PGM2 still has addressability to Space X through TCB A's DU-AL. Because the primary address space has changed, PGM2, however, does not have addressability to Space Y or Space Z. AS2 has its own PASN-AL, which is available for programs that have AS2 as their primary address space. PGM1 and PGM2 can access Space W, a SCOPE=COMMON data space, using the same access list entry token (ALET).

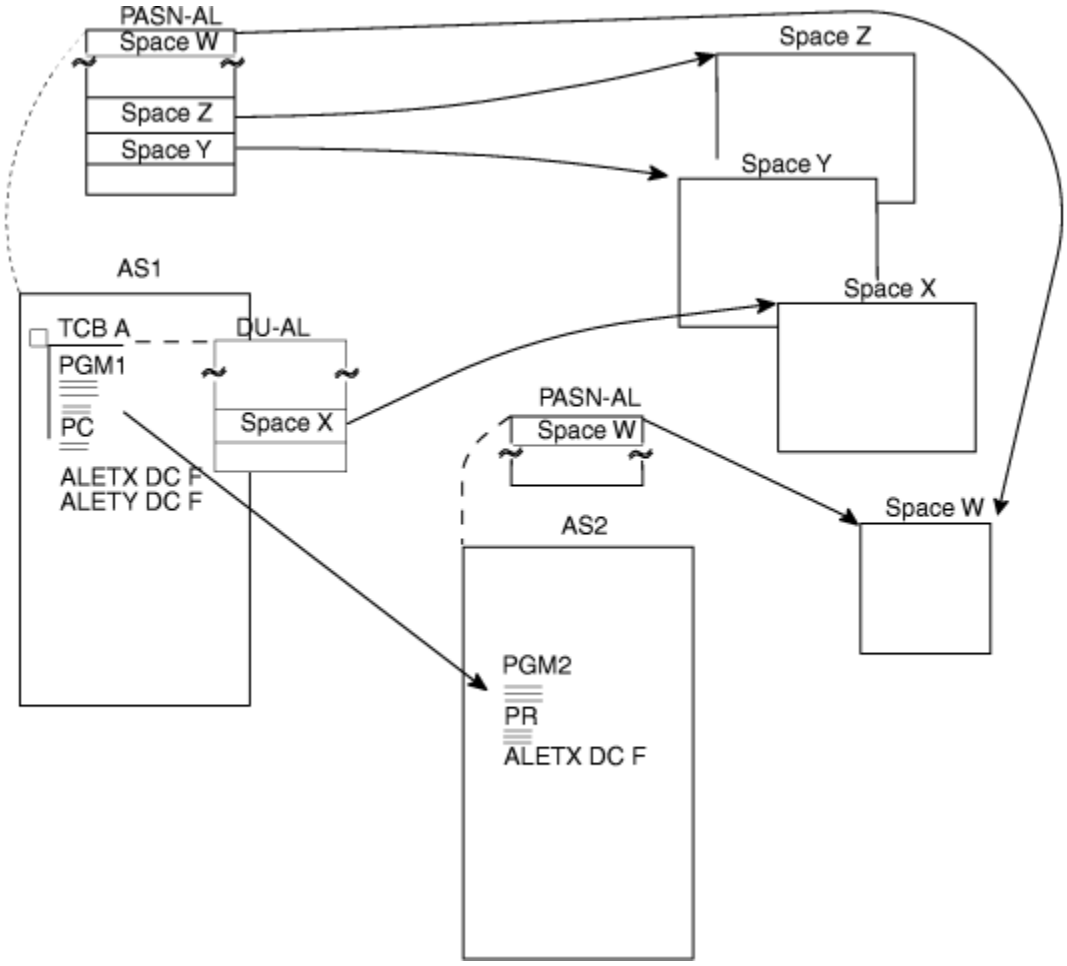


Figure 28. PASN-ALs and DU-ALs at a space switch

Types of access list entries

There are two types of access lists entries for addressability to address spaces. The two types differ from each other in the amount of authority-checking that the system does when a program in AR mode issues a data-referencing instruction and the data is in another address space.

An access list entry for an address space is either a **public entry** or a **private entry**, and a combination of both these types can be on the same DU-AL or PASN-AL.

- A program can access the target address space through a public entry if it has (1) the access list entry that identifies the address space, and (2) the ALET for the entry.
- A program can access the target address space through a private entry if it has (1) an access list entry that identifies the address space, (2) the ALET for the entry, and (3) the appropriate **extended authorization index (EAX)** value.

To be authorized to access the target address space, a program might need a certain EAX value. When it has that value, it is **EAX-authorized** to the address space. Establishing this authorization is a complex programming effort. It is described fully in "EAX-authority to an address space" on page 115.

It is enough at this point to know that:

- Public entries allow any program that has the ALET to use the target address space.
- Private entries can prevent a program from accessing data in an address space.
- Data spaces are accessed only through public entries.

Special ALET values

Each program is provided with three ALETs that allow the program to access its primary, secondary, and home address spaces. You do not need to add an entry to an access list before you use these special ALETs.

Figure 29 on page 103 describes the ALETs that have values of zero, one, and two, and the address spaces they identify.

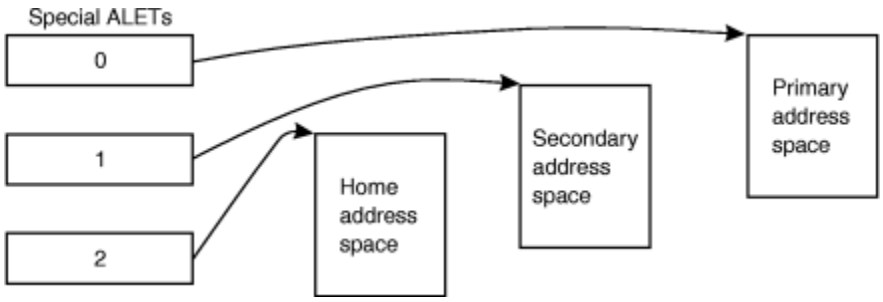


Figure 29. Special ALET values

The three ALETs and examples of their use are:

- An ALET of zero designates the primary address space.

  Some MVS macros require that the issuers have control parameter lists in the primary address space. Use the ALET with the value "0" in the AR that accompanies the GPR containing the address of the parameter list. “Loading the Value of Zero into an AR” on page 104 shows several ways of loading an ALET with the value "0" into an AR.
- An ALET of one designates the secondary address space.

  Programs that are entered through a space-switching PC can reference their caller's parameters (those that reside in the caller's address space) through an ALET of one. For example, instead of using the MVCP and MVCS instructions to move data between primary and secondary, a program can use the MVC instruction and load values of zero (for the primary) and one (for the secondary) in the ARs that are associated with the base registers that the instruction uses. The MVC instruction moves data only between storage areas with appropriate storage keys.
- An ALET of two designates the home address space.

  An ALET of two provides easy access to the home address space while the program is running in AR mode.

  Note: Do not use an ALET of two in a disabled interrupt exit (DIE) routine.

Except for these three special ALET values, a program should never depend on the value of an ALET.

Special ALET Values at a Space Switch

The address space referenced by an ALET of zero changes as the primary address space changes. When a space-switching PC instruction, defined with SASN=OLD, makes a different address space the primary address space, an ALET of zero references the new primary address space. If the PC routine was defined with SASN=NEW, then an ALET of zero and an ALET of one both reference the same address space after a space switch.

The change in the meaning of an ALET of zero and one is important when your program issues a space-switching PC instruction. If you pass an ALET of zero to the routine in the target address space, the zero now refers to that address space. If the parameter is in the address space that the program switched from and the PC was defined SASN=OLD (or used the default), your program must change the value of the ALET from zero to one. An example of this change is in “Example of using TESTART” on page 121.

The change in the meaning of an ALET of zero and one can also be important when your program issues a space-switching PC instruction followed by a non-space-switching PC instruction. For example, if your program issues a PC from address space A to address space B:

- The home address space is A.
- The primary address space is B.
- The secondary address space is A.

If your program then issues a non-space-switching PC instruction within address space B:

- The home address space is still A.
- The primary address space is still B.
- The secondary address space **is now B**.

When you use nested PC routines (one PC routine invokes another PC routine, and that PC routine invokes yet another PC routine, and so on), using the special ALET values is not sufficient to maintain addressability to any address space other than the current primary and secondary address spaces. If you use nested PC routines, use the ALESERV macro to add entries to the DU-AL and pass ALET-qualified addresses. You can use the ALESERV macro with the ADDPASN parameter to add the current primary address space to the DU-AL.

After a program issues a SSAR instruction, an ALET of one references the new secondary address space.

The meaning of the ALET with the value of two (for the home address space) does not change at a space switch.

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 address/data space that contains the data. Therefore, even 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 "0".

The following examples show several ways of placing the value "0" 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 0.

```
            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 code is useful to establish a program's base register GPR and AR from an entry point address contained in register 15. The example assumes that GPR 15 contains the entry point address of the program, PGMA.

```
            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:

```
         SLR   12,12
         SAR   12,12
         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 set up addressability to address/data spaces. Table 11 on page 105 lists some of the functions of the macro, the parameter that provides the function, and the section where the function is described.

Table 11. Functions of the ALESERV Macro

To do the following:	**Use this parameter:**	**Described in this section:**
Add an entry to an access list	ADD	“Adding an entry to an access list” on page 106
Delete an entry from an access list.	DELETE	“Deleting an entry from an access list” on page 113
Add a public entry for the primary address space to a DU-AL.	ADDPASN	“Adding an Entry for the Primary Address Space to the DU-AL” on page 112
Obtain the STOKEN of the current home address space.	EXTRACTH	“Procedures for establishing addressability to an address space” on page 118
Obtain the STOKEN of an address/data space, given the ALET.	EXTRACT	“Obtaining and passing ALETs and STOKENs” on page 108
Find an ALET on an access list, given the STOKEN.	SEARCH	“Adding an entry to an access list” on page 106

You can also find examples of the ALESERV macro in Chapter 6, “Creating and using data spaces,” on page 127.

Setting up addressability to an address/data space

Before your program can use ARs to reference data in an address/data space, it must establish a connection to the address/data space. The important facts to remember about setting up an environment in which your program can use ARs follows:

- Establishing addressability to an address/data space means your program must:
  - Have authority to access data in the address/data space
  - Have an access list entry that points to the address/data space
  - Have the ALET that indexes to the entry

This section describes these actions and gives some examples. The first item in the list, having authority to access data in the address/data space, depends on whether the entry is for a data space or an address space.

- Authority to add an entry for a data space follows certain rules that are summarized in Table 13 on page 132. This table tells what problem state and supervisor state or PSW key 0-7 programs can do with data spaces.

- Authority to add an entry for an address space is determined by whether you require that the system check the EAX value of the program when the program issues an ALESERV ADD macro. CHKEAX=YES asks the system to make sure the program has the appropriate EAX value before executing the macro. CHKEAX=NO tells the system not to check the EAX value of the program. EAX-authority is described in "EAX-authority to an address space" on page 115. Only programs in supervisor state or PSW key 0 - 7 can use CHKEAX=NO. If they have EAX-authorization, problem state programs with PSW key 8 - F can add entries for address spaces to their access lists.

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 another program, or from DSPSERV CREATE, ALESERV EXTRACTH, or ASCRE.
- 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.

Two optional parameters, AL and ACCESS, allow you to limit access to an address/data space:

- AL=WORKUNIT or PASN

  AL specifies the access list, the DU-AL (WORKUNIT parameter) or the PASN-AL (PASN parameter), to which the ALESERV service is to add the entry. The default is WORKUNIT.

  Use AL=WORKUNIT if you want to limit the sharing of the address/data space to programs running under the owning work unit. Use AL=PASN if you want other programs running in the primary address space to have access to the address/data space, or if you are adding an entry for a SCOPE=COMMON data space.
- ACCESS=PUBLIC or PRIVATE

  ACCESS specifies the type of entry, public or private, that the system places on the access list. The default is PUBLIC. Access list entries for data spaces are always PUBLIC.

  Later in this section, you will learn about the EAX-authority that the ACCESS=PRIVATE parameter imposes on the accessing of data in an address space.

The ALESERV ADD process described in this section applies to the data spaces called SCOPE=SINGLE and SCOPE=ALL. For SCOPE=COMMON data spaces, ALESERV ADD adds an entry to all PASN-ALs. "Creating and using SCOPE=COMMON data spaces" on page 145 describes the ALESERV ADD process for these data spaces.

The ALESERV ADDPASN macro adds to the DU-AL an entry for the primary address space. An application would use this macro if its programs run in many address spaces.

ALESERV ADD and ALESERV ADDPASN are the **only** ways to add an entry to an access list. For examples of adding entries to the DU-AL and PASN-AL, see:

- "Example of Adding an Access List Entry for a Data Space" on page 107
- "Examples of establishing addressability to data spaces" on page 108
- "Example of adding an access list entry for an address space" on page 107
- "Adding an Entry for the Primary Address Space to the DU-AL" on page 112.

The example of adding an entry for an address space specifies that the system is not to check for EAX-authority.

If you want to know whether an address/data space already has an entry on an access list, use ALESERV SEARCH. As input to the macro, give the STOKEN of the space, which access list is to be searched, and the location in the list where you want the system to begin to search. If the entry is on the list, the system returns the ALET. If the entry is not on the list, the system returns a code in register 15.

Example of Adding an Access List Entry for a Data Space

The following code uses DSPSERV to create a data space named TEMP. The system returns the STOKEN of the data space in DSPCSTKN and the origin of the data space in DSPCORG. The ALESERV ADD macro adds an entry to a DU-AL and 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.

```
         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 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
DSPCNAME DC   CL8'TEMP    '            DATA SPACE NAME
DSPBLCKS DC   F'1000'                  DATA SPACE SIZE (IN 4K BLOCKS)
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.

A more complete example of manipulating data within this data space appears in “Example of creating, using, and deleting a data space” on page 144.

Example of adding an access list entry for an address space

One way of protecting access to an address space is to require that the system check that the EAX of a program is a certain value. The two times that the system might check this EAX value are when your program tries to

- Add an entry to an access list for the address space
- Access the address space through that access list entry.

If the address space does not need the protection that the EAX offers, a program in supervisor state or PSW key 0 - 7 can use the CHKEAX and ACCESS parameters on ALESERV ADD when it adds the address space entry to the access list:

- CHKEAX=NO tells the system not to check the EAX value of the program that is adding the entry to the access list.
- ACCESS=PUBLIC tells the system not to check the EAX value of a program trying to access data in the address space.

In the following example, a supervisor state or PSW key 0 - 7 program adds an entry to its DU-AL. It asks the system not to check any EAX values, either when it adds the entry or when any instruction accesses data in the address space. The program can have any EAX, including 0. The address space (represented by the STOKEN at location ASTOKENN) is non-swappable. It could have been created by the ASCRE service, which returns a STOKEN at the time of the address space creation.

```
         ALESERV ADD,ALET=ASALET,STOKEN=ASTOKENN,CHKEAX=NO,          X
               AL=WORKUNIT,ACCESS=PUBLIC
         .
*  PROGRAMS CAN NOW ACCESS THE ADDRESS SPACE USING THE ALET, ASALET
         .
ASALET   DS    F                       ALET FOR MVS ADDRESS SPACE
ASTOKENN DS    CL8                     STOKEN FOR MVS ADDRESS SPACE
```

Note that, by using the ALESERV defaults, the program could have issued the following:

```
ALESERV ADD,ALET=ASALET,STOKEN=ASTOKENN,CHKEAX=NO
```

Obtaining and passing ALETs and STOKENs

A program can obtain an ALET through the ALESERV macro with the ADD and ADDPASN parameters. Or, it can receive an ALET from another program.

A program can obtain a STOKEN through DSPSERV CREATE, ALESERV EXTRACT, or ALESERV EXTRACTH. Or, it can receive a STOKEN from another program.

A program can pass an ALET or a STOKEN to another program in the same way it passes other parameter data. MVS has certain rules for passing ALETs, as described in "Rules for passing ALETs" on page 108. It does not have rules for passing STOKENs. However, the ALESERV service determines whether the receiving program can add an entry for the address/data space that a STOKEN represents.

Rules for passing ALETs

To provide addressability to an address/data space, a program might pass an ALET to another program. MVS allows your program to pass the following ALETs:

- An ALET of zero.
- An ALET that indexes into a public entry on a DU-AL, if the program that passes the ALET and the program that receives the ALET run under the same TCB or SRB (that is, they have the same DU-AL).
- An ALET that indexes into the PASN-AL, if the program that passes the ALET and the program that receives the ALET run in the same address space (that is, they have the same PASN-AL).
- An ALET that indexes into the PASN-AL for a SCOPE=COMMON data space.

Do not pass the following ALETs:

- An ALET of one across a space-switching PC linkage. (A space-switching PC instruction changes the program's secondary address space.)
- An ALET that indexes an entry on the PASN-AL, passed to a program in another address space, unless the ALET is for a SCOPE=COMMON data space. Each address space has its own PASN-AL.
- An ALET that indexes an entry on another task's DU-AL. (Each task has its own DU-AL). However, your program can pass such an ALET to a subtask if the subtask was created using the ALCOPY parameter on the ATTACH or ATTACHX macro; the ALET must have been valid at the time of the attach.
- An ALET that indexes a private entry, passed across an interface (through a PC instruction) that changes the EAX. (This rule is described in "EAX-authority to an address space" on page 115.)

"Examples of establishing addressability to data spaces" on page 108 has several examples of programs passing ALETs.

Examples of establishing addressability to data spaces

The best way to describe how to add an access list entry is through examples. This section contains three examples:

- Example 1 sets up addressability to a data space, using the DU-AL. The example continues with a program passing a STOKEN to another program so that both programs can access the data space.
- Example 2 sets up addressability to a data space, using the PASN-AL. This example continues with a program passing an ALET to another program so that both programs can access the data space.
- Example 3 shows how to set up addressability so that two programs in different address spaces can access the same data space.

In these examples, programs share their data spaces with programs running under work units other than their own.

The examples all involve adding entries for data spaces. The reason the examples are not of address spaces is because of the additional decision that you have to make about EAX-authority when you add

entries for address spaces to access lists. Getting EAX-authority is described in "EAX-authority to an address space" on page 115. Turn to that section after you understand how to add entries for data spaces.

Example 1: Getting Addressability Through a DU-AL:

Consider that a supervisor state program named PGM1 created a data space and received a STOKEN from DSPSERV. To add the entry to the DU-AL, PGM1 issues:

```
        ALESERV ADD,STOKEN=STOKDS1,ALET=ALETDS1,AL=WORKUNIT
        .
ALETDS1 DS F
STOKDS1 DS CL8
```

ALESERV accepts the STOKEN, adds an entry to the DU-AL and returns an ALET at location ALETDS1. Figure 30 on page 109 shows PGM1 with the entry for DS1 on its DU-AL. It shows the STOKEN and the ALET.

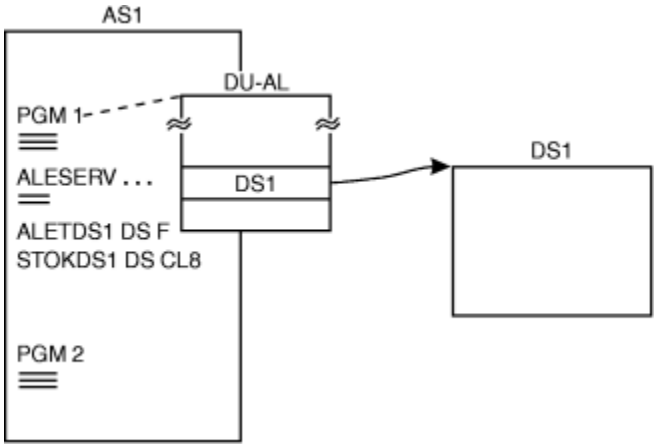


Figure 30. Example 1: Adding an entry to a DU-AL

Consider that PGM2, also in supervisor state, and running under a TCB different from PGM1's TCB, would also like to have access to DS1. PGM1 passes PGM2 the STOKEN for DS1. PGM2 then uses the ALESERV ADD macro to obtain the ALET and add the entry. Figure 31 on page 109 shows PGM2 with addressability to DS1.

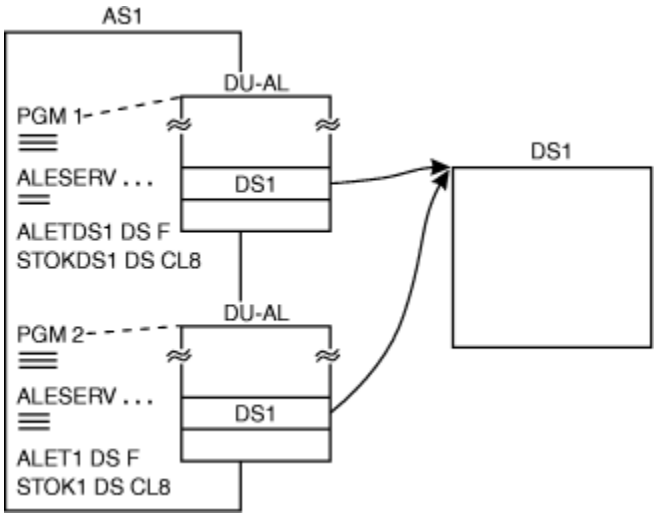


Figure 31. Example 1: Sharing a data space through DU-ALs

Note: A problem state program with PSW key 8 - F can add entries to an access list only for a data space that the program created or owns.

Example 2: Getting Addressability Through a PASN-AL:

In Figure 32 on page 110, consider that PROG1, adds an entry for a data space to the PASN-AL. PROG1 issues the following macro:

```
     ALESERV ADD,STOKEN=STOKDS2,ALET=ALETDS2,AL=PASN
     .
ALETDS2 DS F
STOKDS2 DS CL8
```

ALESERV accepts the STOKEN, adds an entry to the PASN-AL, and returns an ALET at location ALETDS2. Figure 32 on page 110 shows PROG1 with the PASN-AL entry for DS2.

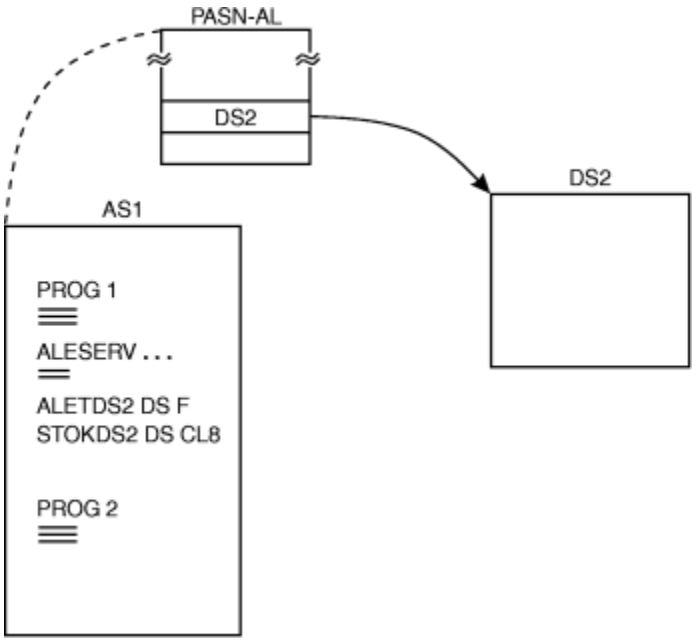


Figure 32. Example 2: Adding an entry to a PASN-AL

Note: A problem state program with PSW key 8-F can add entries to the PASN-AL only for the type of data space called SCOPE=SINGLE.

Consider that PROG2 (either in problem or supervisor state and running under a TCB different from PROG1's) would like to have access to DS2. In this case, both PROG1 and PROG2, because they run in the same address space, share the same PASN-AL. PROG2 does not have to add an entry to its PASN-AL; the entry is already there. PROG1 passes the ALET to PROG2. Figure 33 on page 111 shows that PROG2 has the ALET for DS2 and, therefore, has addressability to DS2 through its PASN-AL.

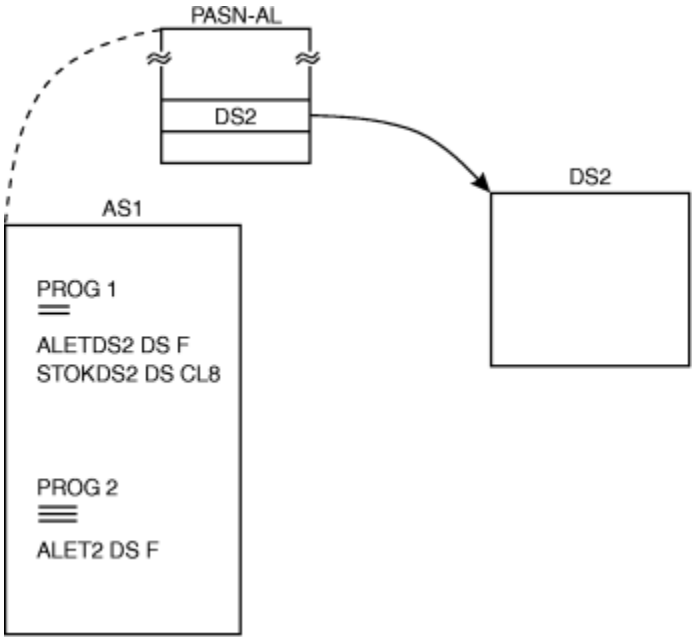


Figure 33. Example 2: Sharing a data space through the PASN-AL

In a similar way, any supervisor state or problem state program that runs in AS1 and has the ALET for DS2 can access DS2.

The SCOPE parameter on DSPSERV determines how the creating program can share the data space. For more information on the SCOPE parameter, see “Scope=single, scope=all, and scope=common data spaces” on page 128.

Example 3: Passing ALETs Across Address Spaces:

Referring to Figure 33 on page 111, consider that PROG1 wants to allow a program in another address space (whose home address space is different from PROG1's) to access data in DS2. Figure 34 on page 112 shows that PROG1 passes the STOKEN for DS2 to PROG2, a supervisor state program in AS2. PROG2 uses the ALESERV macro to add the entry to its DU-AL. PROG2 also could have added the entry to its PASN-AL.

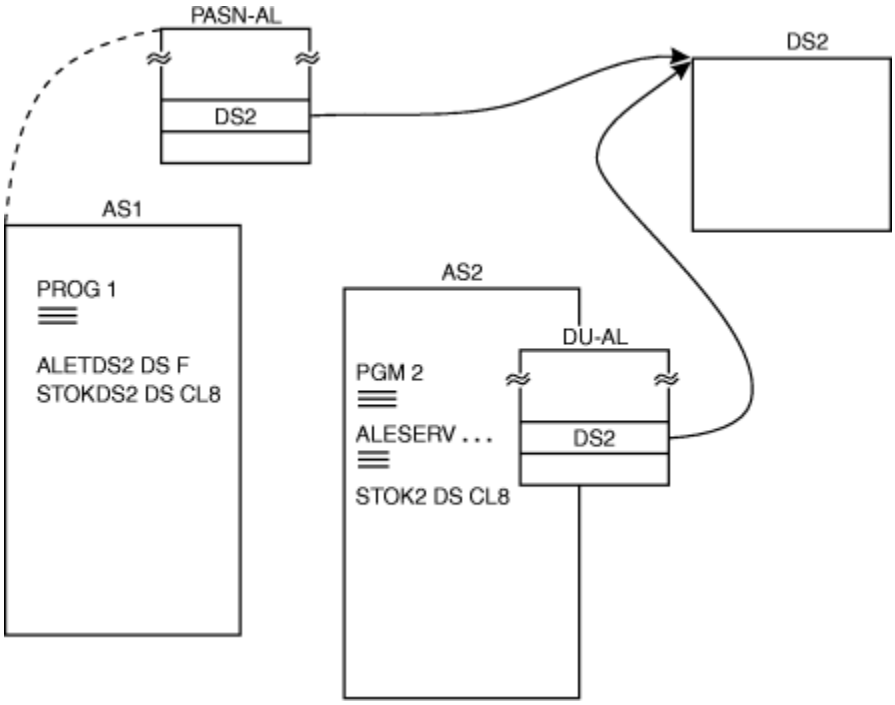


Figure 34. Example 3: Sharing data spaces between two address spaces

Remember that getting addressability to an address space is similar to getting addressability to a data space, with one difference: when you attempt to add an entry for an address space to an access list, ALESERV allows you to require that programs that access the address space through the entry have EAX-authority to the address space.

Adding an Entry for the Primary Address Space to the DU-AL

A program can use the ADDPASN parameter on ALESERV to add an entry for the primary address space to the program's DU-AL. For example, consider the setup in Figure 35 on page 112. TCB A represents PGM1 (in the home address space), PGM2 (currently executing in AS2), PGM3, and PGM4. In other words, all the PC routines shown run under the same TCB with the same DU-AL. PGM3 and PGM4 can use an ALET of 2 to reference the home address space. However, no special ALET exists for PGM3 and PGM4 to reference AS2.

PGM2, without having EAX-authority to the address space, can issue ALESERV ADDPASN to place an entry for AS2 on the DU-AL. This action gives PGM3 and PGM4 addressability to AS2, providing the ALETs are passed to these programs.

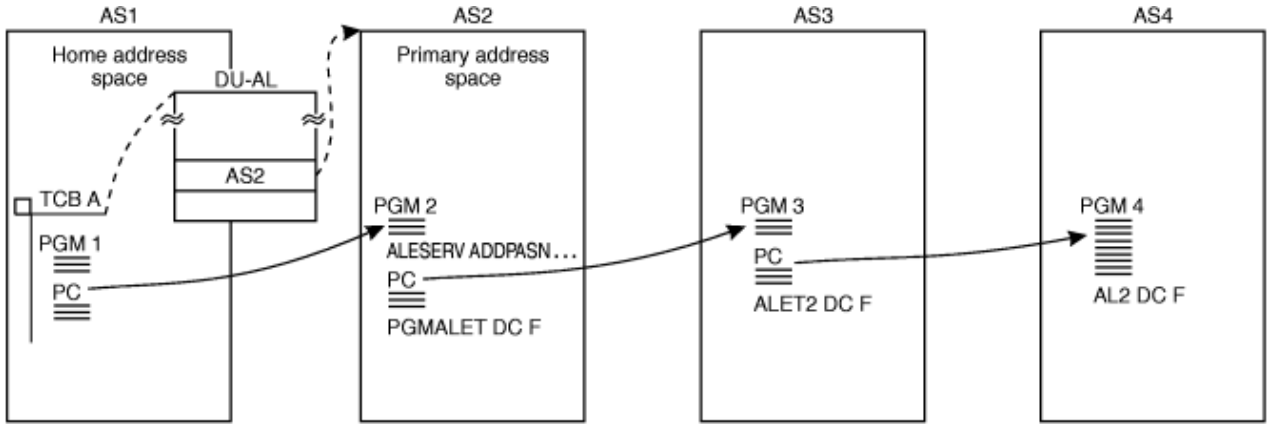


Figure 35. Obtaining the ALET for the Primary Address Space

In the following example, a program adds an entry for the primary address space to the DU-AL as a public entry.

```
ADDPASN  CSECT
ADDPASN  AMODE 31
ADDPASN  RMODE ANY
         BAKR  14,0                    SAVE CALLER'S STATUS ON STACK
         SAC   512                     SWITCH INTO AR MODE
         .
         LAE   12,0                    SET BASE REGISTER AR
         BASR  12,0                    SET BASE REGISTER GR
         USING *,12
         SYSSTATE ASCENV=AR
         .
*  ADD PROGRAM'S PASN AS PUBLIC TO THE PROGRAM'S DU-AL
         ALESERV ADDPASN,ALET=PGMALET
         .
*  BODY OF PROGRAM
         .
*  REMOVE PROGRAM'S PASN FROM DU-AL
         ALESERV DELETE,ALET=PGMALET   REMOVE PASN FROM DU-AL
         .
         PR                            RETURN TO CALLER
         .
PGMALET  DC   F                        ALET FOR PROGRAM'S PASN
         END
```

Using the ALET for the Home Address Space

If a program is part of a subsystem that offers services to many users, it might want to set up addressability for programs executing in other address spaces to reference data in its address space. Figure 36 on page 113 shows a subsystem's home address space as AS1. Setting up addressability to the home address space requires no action. The PC routines that run in address spaces AS2, AS3, and AS4 all run under the same TCB, that of PGM1. Addressability to AS1 is through the special ALET of two. PGM2, PGM3, and PGM4 can place the value 2 in an AR of a AR/GPR pair to reference data in the home address space.

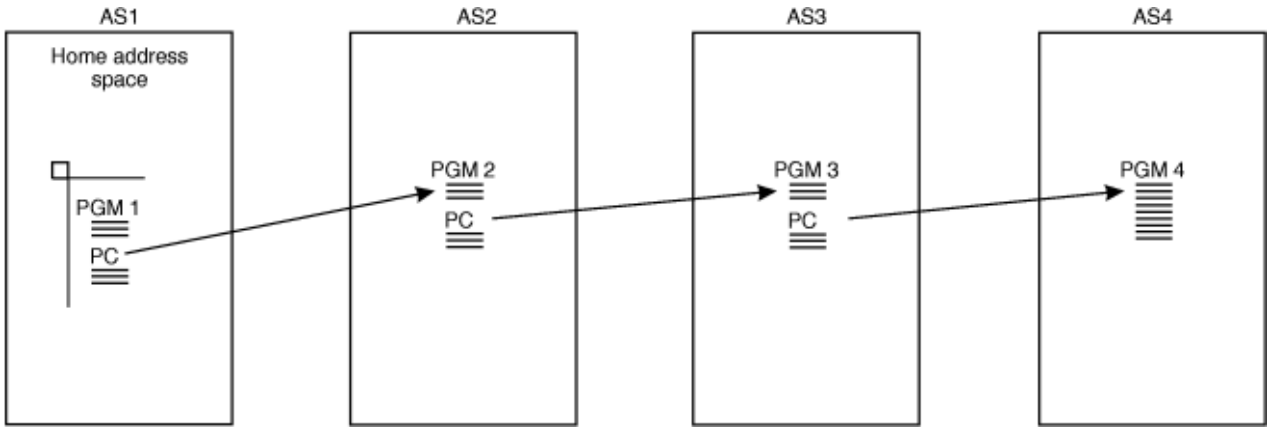


Figure 36. Using the ALET for the Home Address Space

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.

Access lists have a limited size; the DU-AL has 509 entries and the PASN-AL has 510 entries. Therefore, it is a good programming practice to delete entries from an access list when the entries are no longer needed. The specific rules are:

- If a program needs an entry for a short period of time, it should delete the entry when it no longer needs the entry.

- If a program adds an entry and uses that entry during execution, the program does not need to delete the entry; the system deletes the entry when the task terminates.
- If a supervisor state or PSW key 0 - 7 program does not want the system to check the EAX-authority of the program when it deletes an entry for an address space, it should use CHKEAX=NO on ALESERV DELETE. CHKEAX=YES is the default.
- Once the entry is deleted, the system can immediately reuse the ALET.

Programs that share data spaces with other programs have another action to take when they delete an entry from an access list. They should notify the other programs that the entry is no longer connecting the ALET to the data space. Otherwise, those programs might continue to use an ALET for the deleted entry. See "ALET reuse by the system" on page 114 for more information.

Example of deleting a data space entry from an access list

The following example deletes the entry for the ALET at location DSPCALET. The example also includes the deletion of the data space with a STOKEN at location DSPCSTKN.

```
         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.

Example of deleting an address space entry from an access list

The following example deletes the entry that was added in the example in "Example of adding an access list entry for an address space" on page 107.

```
*        .
         ALESERV DELETE,ALET=ASALET,CHKEAX=NO
         .
ASALET   DS    F                         ALET FOR MVS ADDRESS SPACE
```

When ALESERV ADD added the entry to the access list, the system did not check the EAX-authority of the program; in this example, the system does not check the EAX-authority either.

ALET reuse by the system

ALETs are not unique; they index a specific entry on a PASN-AL or DU-AL, connecting a program to an address/data space. When ALESERV DELETE removes an access list entry, the connection between the ALET and the space no longer exists. The access list entry and its corresponding ALET are available for the system to use again. The breaking of the connection and the reuse of the ALET mean that a program using the old ALET:

- Does not gain access to the space
- Might gain access to another space

The system does not check and notify programs about the reuse of an ALET. Therefore, when a program uses ALESERV DELETE to delete an access list entry, the program must ensure that other programs do not use the old ALET.

Consider a program, PROGA, deleting the data space, DSA, and removing the entry from the PASN-AL. The ALET for that entry, ALETA, ceases to have meaning in relationship to DSA. The system, free now to reuse that ALET, assigns ALETA to a new data space, DSB. Suppose other programs in the address space were also using ALETA to access DSA. If PROGA does not tell those programs about the removal of ALETA, those programs will mistakenly access DSB, while intending to access DSA.

This response to the system's removal of the entry and reuse of an ALET is similar to the work a program does after it frees address space storage that it obtained and shared with other programs. When that area

of storage is freed, MVS reuses the area to satisfy a later request for storage. When an access list entry is freed, MVS reuses that ALET to satisfy a later ALESERV ADD request.

EAX-authority to an address space

MVS uses EAXs to control access to address spaces through ARs in a way similar to the way it uses AXs to check if a program has the authority to issue the SSAR instruction with an address space as the target of the SSAR instruction. To be EAX-authorized to the target address space, a program's EAX, when used as an index into the address space's authority table, must point to an entry that indicates SSAR authority. An AX value is related to an address space; all programs running in an address space have the same AX value at any given time. An EAX value is related to a PC routine. The caller has that EAX value only while the PC routine runs. When the PC returns control, the EAX value returns to what it was before the call.

In general, programs start out with an EAX of zero. An EAX of zero is an unauthorized EAX value that can prevent the program from adding or deleting entries for address spaces on its access lists and from accessing data in address spaces other than the one it is running in.

Earlier in the section the two types of access list entries were defined. The definitions are repeated here.

An access list entry for an address space is either a **public entry** or a **private entry**, and a combination of both these types can be on the same DU-AL or PASN-AL. The two types differ from each other by the kind of checking the system does when a program tries to use the entry to access an address space.

- A program can access the target address space through a public entry if it has (1) the ALET for the entry and (2) the access list entry that identifies the address space.
- A program can access the target address space through a private entry if it has (1) the ALET for the entry, (2) an access list entry that identifies the address space and, (3) the appropriate EAX value.

The ACCESS and CHKEAX parameters on ALESERV determine when the system checks the EAX-authority of the program.

- CHKEAX=YES tells the system to check the EAX of the program at the time the program uses the ALESERV macro to add the entry for the address space or delete the entry from the access list.
- ACCESS=PRIVATE tells the system to check the EAX of the program that is attempting to access the target address space through the entry.

It is important that you understand the relationship between the two parameters on ALESERV that determine whether the system checks the EAX value against the SSAR authority in the target address space's authority table. Table 12 on page 115 describes the relationship.

Table 12. Relationship between the CHKEAX and ACCESS parameters on ALESERV

CHKEAX=	ACCESS=	EAX-Checking That Results
YES	PUBLIC	The caller must have EAX-authority to add the entry, but no program needs EAX-authority to access the address space through the entry.
YES	PRIVATE	The caller must have EAX-authority to add the entry for the address space, and all programs that access the address space through that entry must also have EAX-authority.
NO	PUBLIC	The caller does not need EAX-authority to add the entry, and programs that access the address space through the entry do not need EAX-authority. See “Example of adding an access list entry for an address space” on page 107 for an example.
NO	PRIVATE	The caller does not need EAX-authority to add the entry, but programs that access the address space through the entry need EAX-authority.

Once a program places a private entry on the access list (placing the EAX restriction on the users of the address space), a supervisor state or PKM 0 - 7 program running in the address space can use the ATSET macro to turn SSAR authority off. This action means that an EAX, when used as an index into that entry in the authority table, will find SSAR authority turned off. The program with that EAX no longer has EAX-authority to the address space. It is not possible, however, for a program in the target address space to prevent a program from using an entry that was added with the CHKEAX=NO and ACCESS=PUBLIC parameters on ALESERV.

Figure 37 on page 116 gives an example of public and private entries. PGM1 has public entries and private entries on its DU-AL and its PASN-AL. It has the ALETs that allow it to access AS1 and DS1 through its PASN-AL and AS2 and AS3 through its DU-AL.

- **To add the entries for the three address spaces** to an access list, the program might have had to establish EAX-authority to AS1, AS2, and AS3. A supervisor state or PSW key 0 - 7 program can use CHKEAX=NO on ALESERV that allows the program to add the entry, requesting that the system not check its EAX value. Problem programs with PSW key 8 - F must have EAX-authority.
- **To add an entry for the data space** to an access list, the program has to meet certain MVS criteria, as described in Table 13 on page 132.
- **To access data in AS1 and AS2**, the program has to have EAX-authority to those address spaces. To access data in address spaces that have private entries, the system checks the EAX-authority of the program to the address space.
- **To access data in AS3 or the data space**, the program does not need EAX-authority. Entries for data spaces are public entries. To access data in address spaces that have public entries, the system does not check the program's EAX-authority.

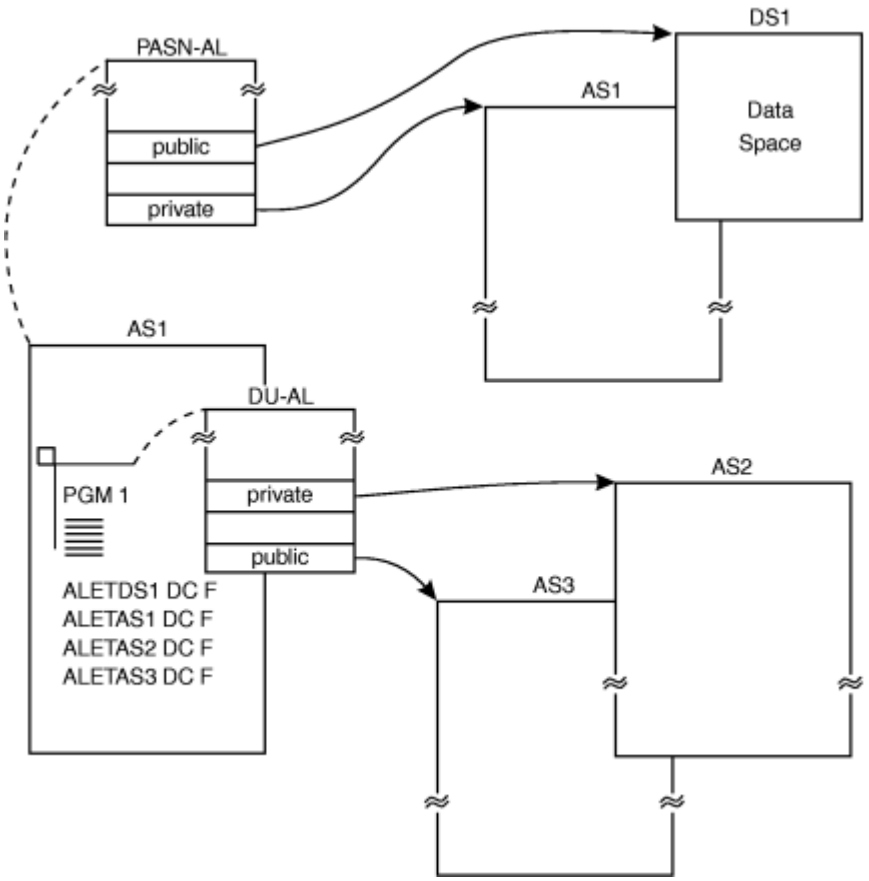


Figure 37. Difference Between Public and Private Entries

To delete the entries for the three address spaces, the program might need EAX-authority to the address spaces. A supervisor state or PSW key 0 - 7 program can use the CHKEAX=YES parameter on the ALESERV macros to require this system checking. Problem programs must have EAX-authority.

Setting the EAX value

The EAX is an index that is similar to the cross memory authorization index (AX), and is obtained in the same manner, through the AXRES macro (see *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* and "Establishing cross memory communication" on page 31 for more information about AXRES). Unlike an AX, which is associated with an address space, an EAX is associated with a PC routine and is available to the programs that call the PC routine.

Figure 38 on page 118 shows the AX value for an address space and the EAX value of a program in supervisor state or PSW key 0 - 7. Assume that the ALESERV ADD macro included CHKEAX=YES, or the default, and that the entry for address space AS2 is a private entry, ACCESS=PRIVATE.

- The **AX value of AS1** indexes into the AT of AS2. The system checks this value on PT or SSAR instructions to find out if a caller in AS1 has the authority to (1) PT to AS2 or (2) set AS2 as its secondary address space. If the entry in AS2's authority table has the PT authority, PGM1 can PT to AS2; if SSAR authority is on, PGM1 can set AS2 as its secondary address space.
- The **EAX value of PCRTN** also indexes into the AT of AS2. The system checks this value when PCRTN uses the ALESERV ADD macro with the CHKEAX=YES option to add an access list entry for the address space to either its DU-AL or its PASN-AL. The system also checks this value when the program tries to reference AS2 through the entry (the entry is a private entry). Because the entry in AS2's authority table indicates SSAR authority, the caller of PCRTN is considered to be EAX-authorized to AS2.

In Figure 38 on page 118, the value of the AX and EAX is 4. The value 4 is an arbitrary value chosen for illustrative purposes. You obtain an AX or EAX value from the AXRES macro. The entry that the AX and EAX indexes into indicates SSAR authority is on, which means that PGM1 is EAX-authorized to the address space.

The example also shows the difference between cross memory data movement with a move to primary (MVCP) and a data movement performed through ARs and the MVC instruction. PGM1 uses the SSAR instruction to establish AS2 as the secondary address space, then it uses MVCP to move data from AS2 to AS1. PCRTN issues the SAC instruction to change the ASC mode to AR mode. Having loaded the addresses and ALETs into the AR/GPR correctly, PCRTN uses MVC to move data from AS2 to AS1.

Note:

1. If PCRTN had used CHKEAX=NO on ALESERV, the system would not have checked the EAX.
2. If PCRTN had used ACCESS=PUBLIC on ALESERV, the system would not have checked the EAX value when programs referenced that address space through that access list entry.
3. Consider the storage key and data access and integrity issues when you add entries for address spaces. Most problem state programs execute with a PSW key of 8, which allows them to use public access list entries to modify data in storage that has storage key 8.
4. The EAX value can be the same as the AX value.

In some cases, supervisor state or PSW key 0 - 7 programs in the target address space can change the EAX checking that the system does for programs accessing data in their address space. For example, in Figure 38 on page 118, a program in AS2 could use the ATSET macro to change SSAR authority in the fifth entry in the authority table. Because the entry was added as CHKEAX=YES and ACCESS=PRIVATE, if the program turned SSAR authority off, PCRTN could no longer access the address space through that access list entry. If the entry had been added CHKEAX=NO and ACCESS=PUBLIC, programs in the target address space would be unable to prevent access through those access list entries.

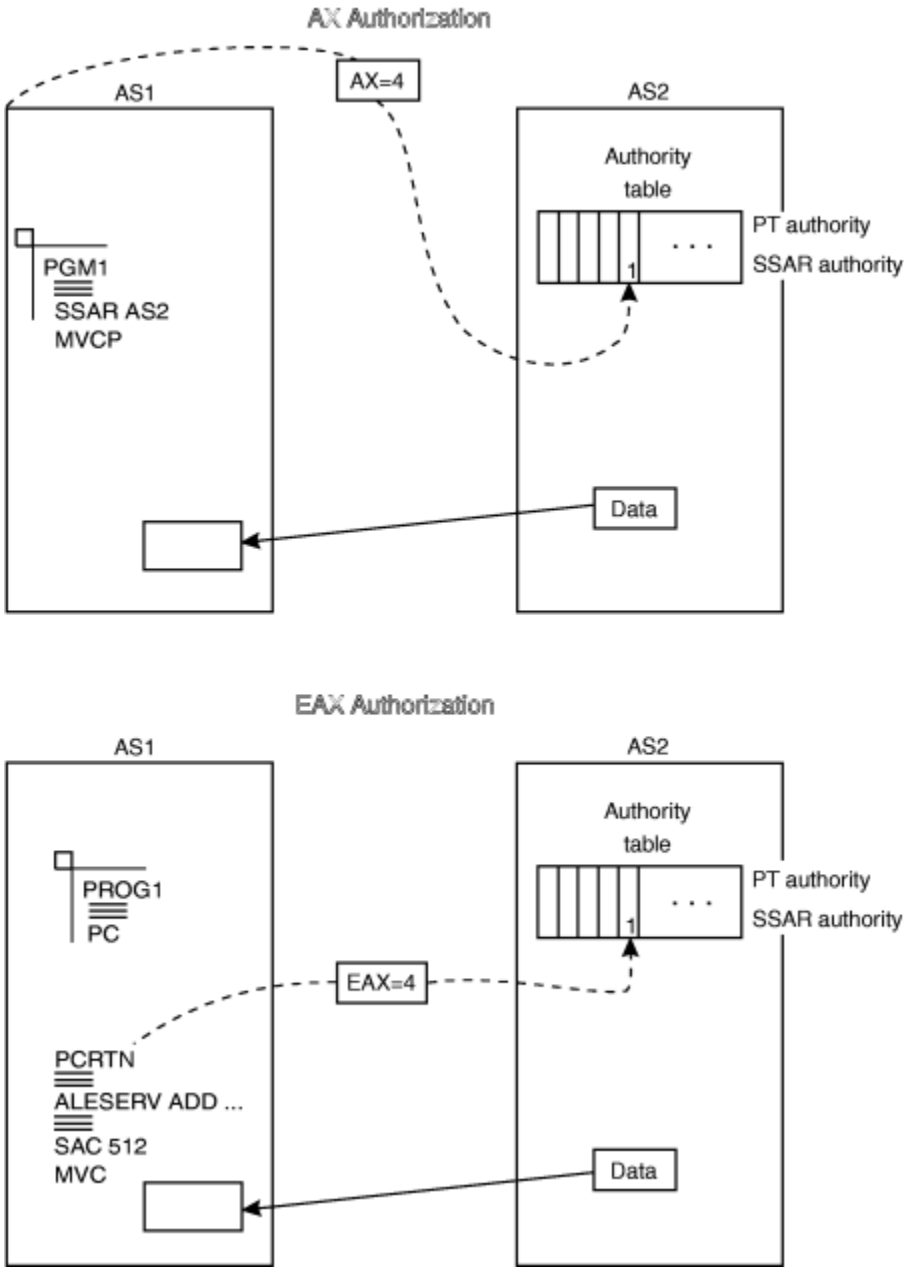


Figure 38. Comparison of an AX and an EAX

Procedures for establishing addressability to an address space

To establish the environment in which a program can add an access list entry for an address space and access data through private entries, programs in the accessing address space and the target address space must:

- Place the entry in an access list
- Obtain the ALET for the entry
- Issue a stacking PC; programs must define the stacking PC through the ETDEF macro, and use the EAX parameter on ETDEF to specify the appropriate EAX value.

  Issuing the stacking PC means that you must establish some of the same linkages that are described in Chapter 3, “Synchronous cross memory communication,” on page 19 and have a program in the target address space help in establishing those linkages.

Remember that the EAX-authority to an address space is the same as the SSAR authority. In other words, authority for an address space to issue the SSAR instruction for the target address space is the same as the authority to add an entry for an address space to an access list or access data in that address space through ARs. The same ATSET macro that sets the PT and SSAR authority in the target address space's authority table also sets authority table bits that correspond to the EAX.

As you read the following procedures for a program in the accessing address space and a program in the target address space, keep three facts in mind:

- A problem state program with PSW key 8 - F must be EAX-authorized to the target address space before it can issue the ALESERV ADD or ALESERV DELETE macros for that address space. A supervisor state or PSW key 0 - 7 program might require EAX-authorization.
- The target address space must give explicit permission to the accessing program.
- The only way for a program in the accessing address space to get a nonzero EAX is to issue a stacking PC instruction, which establishes the appropriate EAX.

Procedures for the accessing address space

The procedures for the program in the accessing address space include writing a stacking PC routine (Step 1), establishing the environment in which it can be called (Steps 2 through 6), and invoking the PC routine (Step 7). The procedures are as follows:

1. Write a PC routine that will run in the accessing address space.

   The PC routine contains the ALESERV ADD request for an address space and any code that manipulates data in the target address space. (See Step 7.)
2. Issue the AXRES macro to reserve an AX to be placed in the entry table descriptor (ETD) as the EAX value of the PC routine.
3. Place the AX in a common area so that a program in the target address space can obtain it and set its AT accordingly.
4. Issue the ETDEF macro to build the PC routine's ETD.

   On the EAX parameter, code the address of the AX value that is to be the EAX of the PC routine.
5. Establish the cross memory structures in the accessing address space so that the stacking PC routine can be called.
   - Issue the ETCRE macro to build the entry table.
   - Issue the LXRES macro to reserve a linkage index (LX) in the linkage table.
   - Issue the ETCON macro to connect the entry table to the linkage table entry.
6. Wait (using the WAIT macro) for the program in the target address space to set its authority table entry.

   The program in the target address space will accept the EAX value and set the authority table accordingly. In this way, the target address space "gives permission" to the accessing program to reference data in the address space.
7. Invoke the stacking PC routine to gain EAX-authorization.

   While the PC routine is running, the caller has the EAX value that the EAX parameter on ETDEF defined. The PC routine can perform the following actions:
   - Issue the ALESERV ADD macro to add an entry for the target address space to the access list.
   - Manipulate data in the target address space, if needed.

     If the routine uses the ALET that ALESERV returns in accessing or manipulating data in the target address space, the PC routine must be in AR mode (either through the SAC instruction or through the AR parameter on the ETDEF macro that defined the PC routine).
   - Use the PR instruction to return to the caller and restore the EAX value that existed before the PC routine ran.

Procedures for the target address space

The procedure for a program in the target address space is as follows:

1. Wait (using the WAIT macro) for the accessing address space to place the AX (to be used as the EAX) into the common area.
2. Change to supervisor state to invoke the cross memory macros.
3. Retrieve the AX value that the accessing address space passed and issue the ATSET macro to set the authority table.
4. Notify (using the POST macro) the accessing address space that the authority table is set.

   This action tells the accessing address space that a program can successfully issue the ALESERV macro.

Changing an EAX value

Two instructions can change a program's EAX value:

- The PC instruction, providing the value on the EAX parameter changes the EAX value for the caller
- The PR instruction, which restores the EAX value to the value that the caller had before it entered the PC routine.

The program has that EAX value only while the PC routine is running.

Freeing an EAX value

When you no longer need an EAX value, you should use the AXFRE macro to return the EAX to the system. Before you issue AXFRE, make sure that the EAXs being returned are no longer being used by any address space, or your program is abnormally terminated.

Checking the authority of callers

A PC routine might want to check the validity of the ALETs that a calling program passed and also check the EAX-authority of the calling program. Making such checks is a good programming practice for PC routines that change the EAX value and space-switching PC routines.

The TESTART macro tests the validity of an ALET and the EAX-authority of the caller to access the address/data space that the ALET represents. The macro returns a code that identifies whether the ALET is:

- 0
- A valid ALET for the DU-AL
- A valid ALET for the PASN-AL
- 1
- Invalid

Input to TESTART is the ALET that it received and the EAX of the calling program. To get the EAX, a program can issue the extract stacked state (ESTA) instruction to retrieve the EAX from the current linkage stack entry. The first information field in the linkage stack entry contains the EAX of the caller. (If the EAX is 0, the ALET is for a public entry.) See “Extract stacked state (ESTA) instruction” on page 15 for a description of ESTA instruction, an example of its use, and the format of the information field. Figure 39 on page 121 shows an example of PGM1 (in problem state) requesting service from PCRTN (in supervisor state).

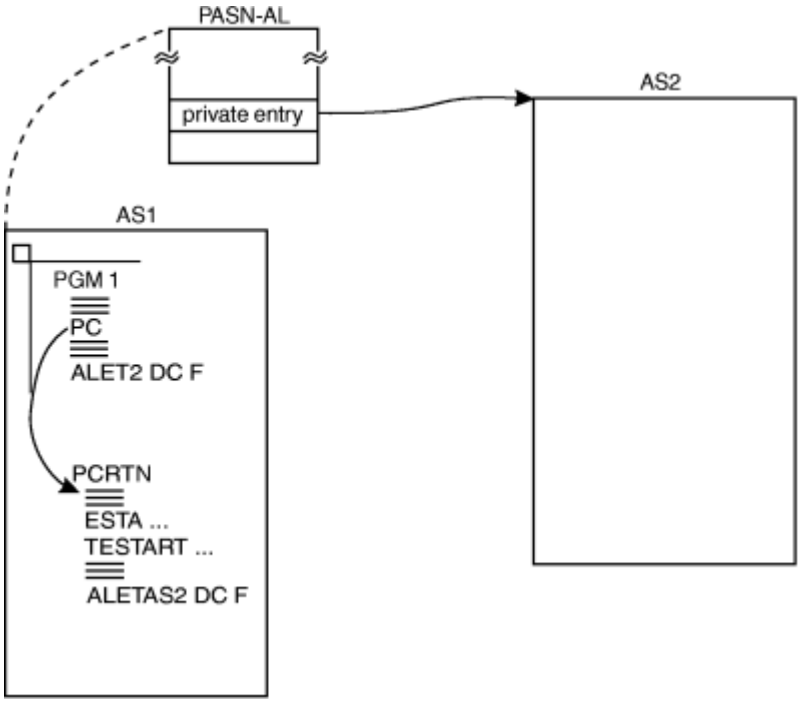


Figure 39. Checking the Validity of an ALET

Having received the ALET from another program, PGM1 passes the ALET to PCRTN for PCRTN's use. Before it uses the ALET, PCRTN issues the TESTART macro to test its validity.

To get the EAX value of the caller, PCRTN issues the ESTA instruction. PCRTN determines which of the return codes is acceptable for its purposes. For example, it might accept only ALETs that index public entries on a DU-AL.

Example of using TESTART

In the following example, the stacking PC routine validates the caller's ALET to reduce the probability of taking a program check referencing storage. Only ALETs of 0 or ALETs on the DU-AL are valid for a space-switch PC routine. Because the primary and secondary address spaces have changed at the space-switch, an ALET of 1 or an ALET on the PASN-AL are not valid.

The following code checks for ALETs of 0 and 1. It then changes the ALET of 0 to 1 to reflect the change in primary and secondary address space at the space switch.

```
         SLR    5,5    SELECT ESTA CODE FOR DESIRED INFORMATION
*                          0 - PKM / SASN / EAX / PASN
         ESTA   4,5    LOAD INFORMATION INTO GPRs 4 AND 5
*                          GPR5 NOW CONTAINS EAX AND PASN
         EREG   1,1    GET CALLER'S AR/GPR1 FROM LINKAGE STACK
         .
         TESTART ALET=(1),EAX=(5)
         .
* THESE ARE THE       rc=0 - ALET IS 0
* TESTART RETURN      rc=4 - ALET IS VALID DU-AL ALET
* CODES.  0 AND 4     rc=8 - ALET IS VALID PASN-AL ALET
* MEANS A GOOD        rc=12 - ALET IS 1
* ALET.               rc=16 - ALET IS NOT VALID
*                     rc=20 - UNEXPECTED ERROR
         .
         LA     0,4
         CR     15,0      If RC=4, VALID DU-AL ALET
         BE     ALETOK
         LTR    15,15     If RC=0, ALET IS 0 (PRIMARY)
         BNZ    BADALET   OTHERWISE, ALET IS NOT USEABLE.
         .
         LA     0,1       BECAUSE ALET IS 0, MUST CHANGE IT TO 1
         SAR    1,0       DUE TO THE SPACE-SWITCH
ALETOK   EQU    *
         MVC    PARM,8(1) COPY USER PARAMETERS INTO LOCAL STORAGE
```

```
BADALET  EQU    *
         .
         .
PARM     DS     CL8       COPY OF USER PARAMETERS
         .
```

Obtaining storage outside the primary address space

The STORAGE macro allows you to obtain storage in an address space other than the primary address space. The program must be in primary mode or AR mode and in PSW key 0 - 7 or supervisor state. The ALET parameter on the STORAGE macro identifies the address space.

The following example shows how a program can obtain storage in another address space, provided it has the proper authorization. The caller uses ALESERV ADD to obtain an ALET representing the address space (or uses an ALET with the value 1 or 2) and then STORAGE OBTAIN to obtain storage. The example assumes that the caller passes the STOKEN of the target address space by a pointer in AR/GPR 1 on entry. It also assumes that this program has the proper authorization to the target address space.

The program requests one page (4096 bytes) of storage above 16 megabytes in subpool 0 in the target address space.

```
STOR2    CSECT
STOR2    AMODE 31
STOR2    RMODE ANY
         .
*  ENTRY LINKAGE
         .
         BAKR  14,0               SAVE CALLER'S STATUS ON LINKAGE STACK
         SAC   512                SWITCH INTO AR MODE
         SYSSTATE ASCENV=AR       SET GLOBAL BIT INDICATING AR MODE
         LAE   12,0(15,0)         ESTABLISH ADDRESSABILITY
         USING STOR2,12
         STORAGE OBTAIN,LENGTH=72 GET STANDARD SAVE AREA
         LAE   13,0(1,0)          SET UP SAVE AREA POINTER
         MVC   4(4,13),=C'F1SA'   INDICATE IN SAVE AREA THAT CALLER'S
*                                 STATUS IS ON THE LINKAGE STACK
         .
         EREG  R1,R1              RESTORE CALLER'S PARAMETER REGISTER
         USING PARMLIST,R1        ESTABLISH ADDRESSABILITY TO
*                                 PARAMETER LIST
         MVC   ASTOKEN,CSTOKEN    COPY STOKEN INTO LOCAL STORAGE
         DROP  R1                 DROP BASING REGISTER
         .
*   ADD THE ADDRESS SPACE REPRESENTED BY THE TARGET STOKEN TO
*   THE DU-AL AS A PUBLIC ENTRY.
         .
         ALESERV ADD,STOKEN=ASTOKEN,AL=WORKUNIT,ACCESS=PUBLIC,         X
               ALET=ASALET
         .
```

```
*   NOW OBTAIN ONE PAGE (4096 BYTES) OF STORAGE IN SUBPOOL 0 IN THAT
*   ADDRESS SPACE,  ABOVE 16MB.
         .
         STORAGE OBTAIN,LENGTH=4096,SP=0,ALET=ASALET,LOC=ANY,ADDR=ASADDR
         .
```

```
*   SET UP REGISTERS TO POINT AT THE STORAGE JUST OBTAINED.
         .
         L     R4,ASADDR    LOAD ADDRESS OF STORAGE INTO GPR4
         L     R3,ASALET    LOAD ALET OF STORAGE INTO GPR3
         SAR   R4,R3        LOAD ALET INTO AR4
         .
*     AR/GPR 4 CAN NOW BE USED TO REFERENCE THE STORAGE IN
*     THE OTHER ADDRESS SPACE.
         .
         USING ARSTOR,R4
         MVC   FIELD1,DATA1    MOVE DATA INTO THE ADDRESS SPACE
         MVC   FIELD2,DATA2
         DROP  R4
         .
```

```
*   RELEASE THE STORAGE PREVIOUSLY OBTAINED.
         .
```

```
         STORAGE RELEASE,LENGTH=4096,ALET=ASALET,ADDR=ASADDR,SP=0
         .
*   REMOVE THE ENTRY FROM OUR ACCESS LIST
         .
         ALESERV DELETE,ALET=ASALET
         .
```

```
*  EXIT LINKAGE
         .
         LAE   1,0(13,0)          GET ADDRESS OF SAVE AREA
         STORAGE RELEASE,ADDR=(1),LENGTH=72  RELEASE THE SAVE AREA
         SLR   15,15              SET A RETURN CODE OF ZERO
         PR                       RETURN TO CALLER
         .
```

```
*    VARIABLES AND REGISTERS
         .
ASTOKEN  DS    CL8           STOKEN OF ADDRESS SPACE
ASADDR   DS    F             ADDRESS OF STORAGE IN ADDRESS SPACE
ASALET   DS    F             ALET REPRESENTING ADDRESS SPACE
DATA1    DC    CL4'BLUE'
DATA2    DC    CL4'PINK'
         LTORG
         .
R1       EQU   1
R3       EQU   3
R4       EQU   4
         .
```

```
*    PARAMETER LIST MAPPING
         .
PARMLIST DSECT
CSTOKEN  DS    CL8           USER'S STOKEN
         .
*    MAPPING OF STORAGE IN TARGET ADDRESS SPACE
         .
ARSTOR   DSECT
FIELD1   DS    CL4           Area 1
FIELD2   DS    CL4           Area 2
         END
```

What access lists can an asynchronous exit routine use?

If your program issues a macro that causes an asynchronous exit routine to run, that routine cannot use the DU-AL associated with your program. The system gives the routine an empty DU-AL for its own use and an EAX value of zero. The routine can use the PASN-AL associated with the primary address space.

Such asynchronous exit routines include those caused by the ATTACH macro with the ETXR parameter, the STIMER macro with the EXIT parameter, the SCHEDXIT macro, and some attention and I/O exit routines.

When control returns to your program from an asynchronous exit routine, the system deletes the DU-AL associated with the asynchronous routine and restores your program's DU-AL and EAX value.

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 one of the following:
  - *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*

  - *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
  - *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
  - *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*
  - *z/OS MVS Programming: Assembler Services Reference ABE-HSP*
  - *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

  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 name of the macro for the AR mode caller is the same as the name of the macro for primary mode callers, except the macro that supports the AR mode caller ends with an "X". This document 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. For these exceptions, check the macro descriptions in one of the following:

  - *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
  - *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
  - *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
  - *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*
  - *z/OS MVS Programming: Assembler Services Reference ABE-HSP*
  - *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.
- 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 that you always use ESTAEX unless your program and your recovery routine are in 24-bit addressing mode, or your program requires a branch entry. In those cases you should use ESTAE.

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.

If your program must issue macros while it is in AR mode, make sure the macros support AR mode callers and that SYSSTATE ASCENV=AR is coded. For information about macros that support AR mode callers and how to issue the macros correctly, see "Address Space Control (ASC) Mode" in the appropriate macro description in one of the following:

- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

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.

Passing parameters to MVS macros in AR mode

The rules for passing ALETs to MVS macros are similar to the rules for passing ALETs to programs. For programs in AR mode, the system allows you to pass the following ALETs:

- An ALET with the value of zero, signifying that the parameter data resides in the caller's primary address space
- An ALET that indexes to a public entry on the caller's DU-AL.

Do not pass other ALETs; the system does not support them.

Some of the macros that support callers in AR mode require that parameter lists be in the primary address space. To learn where the input parameter lists must reside, see the macro descriptions in one of the following:

- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*
- *z/OS MVS Programming: Assembler Services Reference ABE-HSP*.

Some macro services accept control parameters from a program. Do not pass a parameter that resides at location zero in a data space to a macro service. Some macros use the value 0 to designate that a parameter list was not specified.

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 *z/OS MVS IPCS Commands* for more information about the ARCHECK subcommand.

Chapter 6. Creating and using data spaces

A **data space** is a range of up to two gigabytes of contiguous virtual storage addresses that a program can directly manipulate through assembler instructions. Unlike an address space, a data space contains only data; it does not contain common areas or system data or programs. Program code does not execute in a data space, although a program can reside in a data space as nonexecutable code.

The DSPSERV macro with the TYPE=BASIC parameter (the default) manages data spaces. Use this macro 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 from central storage

A program's ability to create, delete, and access data spaces depends on whether it is a problem state program with PSW key 8 - F, a supervisor state program, or a PSW key 0-7 program. All programs can create, access, and delete the data spaces they own or created, and can share their data spaces with other programs running in the same address space. In addition, supervisor state or PSW key 0-7 programs can share their data spaces with programs in other address spaces. ***Unless otherwise stated, this section describes what the supervisor state or PSW key 0-7 programs can do.***

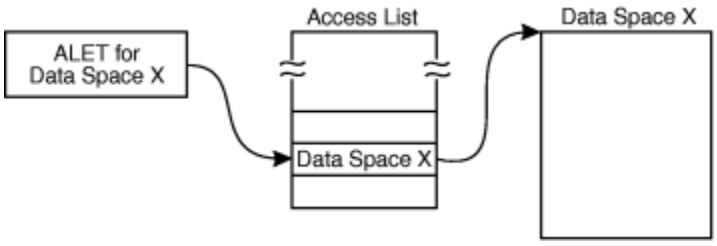
Use this section to help you create, use, and delete data spaces. In addition, four sources of information can help you understand how to use data spaces:

- Chapter 1, “An introduction to extended addressability,” on page 1 can help you verify that a data space, rather than a hiperspace would be the best choice for your program. See “Basic decision: data space or hiperspace” on page 5.
- Chapter 5, “Using access registers,” on page 93, contains many examples of setting up addressability to data spaces.
- One of the following contains the syntax and parameter descriptions for the macros that are mentioned in this chapter:
  - *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
  - *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
  - *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
  - *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.
- *Principles of Operation* contains descriptions of how to use the instructions that manipulate access registers.

Referencing data in a data space

To reference the data in a data space, the program must be in access register (AR) mode. Assembler instructions (such as load, store, add, and move character) move data in and out of a data space and manipulate data within it. Assembler instructions can also perform arithmetic operations on the data.

When a program uses the DSPSERV macro to create a data space, the system returns a **STOKEN** that uniquely identifies the data space. (Data spaces do not have ASIDs.) The program then gains access to the data space: it uses the ALESERV macro to add an entry to an access list and obtain an access list entry token (ALET). The entry on the access list identifies the newly created data space and the ALET indexes the entry.



The process of giving the STOKEN to ALESERV, adding an entry to an access list, and receiving an ALET is called **establishing addressability to the data space**. The access list can be one of two types:

- A dispatchable unit access list (DU-AL) — the access list that is associated with a TCB or SRB
- A primary address space access list (PASN-AL) — the access list that is associated with an address space

Relationship between the data space and its owner

Your program can create a data space, but it cannot own the data space. If the unit of work that represents the program is a TCB, that TCB is the **owner** of the data space, unless the program assigns ownership to another TCB. If the unit of work is an SRB, the program **must** assign ownership to a TCB. Because of this assignment of ownership, the owner of the data space and the creator of the data space are not always the same TCB.

The data space virtual area is available to programs that run under the TCB that owns the data space and is available, in some cases, to other programs.

When a TCB terminates, the system deletes any data spaces that the TCB owns. The system swaps a data space in and out as it swaps in and out the address space that dispatched the owning TCB. Thus, data spaces shared by programs that run in other address spaces **must** be owned by TCBs in non-swappable address spaces.

A data space can remain active even after the creating TCB terminates. When the program creates a data space, it can assign ownership of the data space to a TCB that will outlast the creating TCB. In this case, the termination of the creating TCB does not affect the data space.

Because access lists and data spaces belong to units of work, keep in mind the relationship between the program and the unit of work under which it runs. For simplicity, however, this chapter describes access lists and data spaces as if they belong to programs. For example, "a program's DU-AL" means "the DU-AL that belongs to the TCB under which a program is running".

Scope=single, scope=all, and scope=common data spaces

Data spaces are either SCOPE=SINGLE, SCOPE=ALL, or SCOPE=COMMON, named after the SCOPE parameter on the DSPSERV CREATE macro.

- **SCOPE=SINGLE** data spaces

  A SCOPE=SINGLE data space with an entry on a PASN-AL can be used by programs running in the owner's address space. A SCOPE=SINGLE data space with an entry on a DU-AL can be used by programs represented by TCBs or SRBs whose home address space is the same as the owning TCB. It would typically be used in ways similar to private storage in an address space.

- **SCOPE=ALL** data spaces

  A SCOPE=ALL data space can be used by programs running in the owner's primary address space and other address spaces. SCOPE=ALL data spaces provide a way to share data selectively among programs running in multiple address spaces.

- **SCOPE=COMMON** data spaces

  A SCOPE=COMMON data space can be used by all programs in the system. It provides a commonly addressable area similar to the common service area (CSA). A SCOPE=COMMON data space is sometimes called a common area data space.

The home address space of the owner of a SCOPE=ALL or SCOPE=COMMON data space must be non-swappable during the time that other address spaces have access to the data space.

Rules for creating, deleting, and using data spaces

To protect data spaces from unauthorized use, the system uses certain rules to determine whether a program can create or delete a data space or whether it can access data in a data space. The rules for problem state programs with PSW key 8 through F differ from the rules for programs that are supervisor state or PSW key 0 through 7. The table on page Table 13 on page 132 summarizes these rules and the example in Figure 40 on page 131 illustrates them.

A program in **supervisor state or PSW key 0-7** can:

- **Create** a data space if its home or primary address space is the same as the intended owner's home address space.
- **Delete** a data space if its primary or home address space is the same as the owner's home address space.
- **Release an area** of a SCOPE=SINGLE data space if its primary or home address space is the same as the owner's home address space and its PSW key is the same as the storage key of the data space. It can release an area of a SCOPE=ALL or SCOPE=COMMON data space if its PSW key is zero or matches the storage key of the data space.
- **Extend the current size** of any data space.
- **Page in and out of central storage** the storage of any data space.
- **Establish addressability** to a data space through the ALESERV macro (if the program does not already have an entry on its DU-AL or a PASN-AL) and obtain the ALET that indexes the entry. When it adds an entry, the program can specify whether it wants the entry on its DU-AL or the PASN-AL. A program can add entries:
  - For a SCOPE=SINGLE data space to its DU-AL, if its home address space is the same as the owner's home address space
  - For a SCOPE=SINGLE data space to its PASN-AL, if the PASN-AL belongs to the same address space as the owner
  - For any SCOPE=ALL data space to its DU-AL and its PASN-AL
  - For any SCOPE=COMMON data space to its PASN-AL

  On the ALESERV macro, you can take the default for the ACCESS parameter. All access list entries for data spaces are public (ACCESS=PUBLIC). A public entry allows a program to access data in a data space, without having to establish EAX-authority. See “Types of access list entries” on page 102 for more information about public entries.

  Note that problem state programs with PSW key 8 - F can add entries to their PASN-ALs for the SCOPE=SINGLE data spaces they own or created. Supervisor state or PSW key 0-7 programs can add entries on behalf of problem state programs and pass ALETs to the problem state programs.
- **Access data** in a data space

  Once an entry for the data space is on its DU-AL, a program having the ALET for the entry can access the data space. Once an entry for the data space is on the PASN-AL, all programs running with that PASN-AL and having the ALET can access the data space. Note that data space storage is also subject to storage key and fetch protection.

  A program can attach a subtask and pass a copy of its DU-AL to the subtask. This action allows the program and the subtask to share the data spaces that have entries on the DU-AL at the time of the attach.

Example of the rules for accessing data spaces

Another way of describing the rules for accessing data spaces is through an example. Figure 40 on page 131 shows two address spaces and two data spaces. The entries in the PASN-AL and DU-AL are identified.

Two programs run in address space AS1, both of which own data spaces:

- A problem state program, PGM1, running under TCB A that owns SCOPE=SINGLE data space DS1

  PGM1 can access DS1 through the DU-AL, because it runs in problem state. PGM1 cannot add an entry for DS1 to the PASN-AL.
- A supervisor state program, PGM2, running under TCB B that owns SCOPE=ALL data space DS2

  PGM2 can access DS2 through its PASN-AL. (If PGM1 passes the STOKEN for DS1 to PGM2, PGM2 could add an entry to the PASN-AL for DS1. If PGM2 passes the ALET for DS2 to PGM1, PGm1 could access DS2 through the PASN-AL.)

Two programs run in address space AS2, neither of which own data spaces:

- A problem state program, PGM3, running under TCB C

  PGM3 cannot access either DS1 or DS2.
- A supervisor state program, PGM4, running under TCB D

  PGM4 can access DS2 through its DU-AL.

PGM2 has passed a STOKEN for the SCOPE=ALL data space DS2 to PGM4 in address space AS2. PGM4 used the STOKEN as input to ALESERV, which placed an entry for DS2 on the DU-AL and returned the ALET. PGM4 could have added the entry for DS2 to its PASN-AL.

Earlier in this chapter, it was stated that storage within a data space is available to programs that run under the TCB that owns the data space. The exception to this statement is when the owning TCB has the data space entry on the PASN-AL and a program running under the TCB uses a space-switching PC instruction. During the time that the primary address space is not the owning TCB's home address space, the program cannot access the data space. For example, in Figure 40 on page 131, consider what happens to PGM2 if it should PC to PGM3. Because the entry for DS2 is on AS1's PASN-AL, PGM2 cannot access DS2 while it is running in AS2.

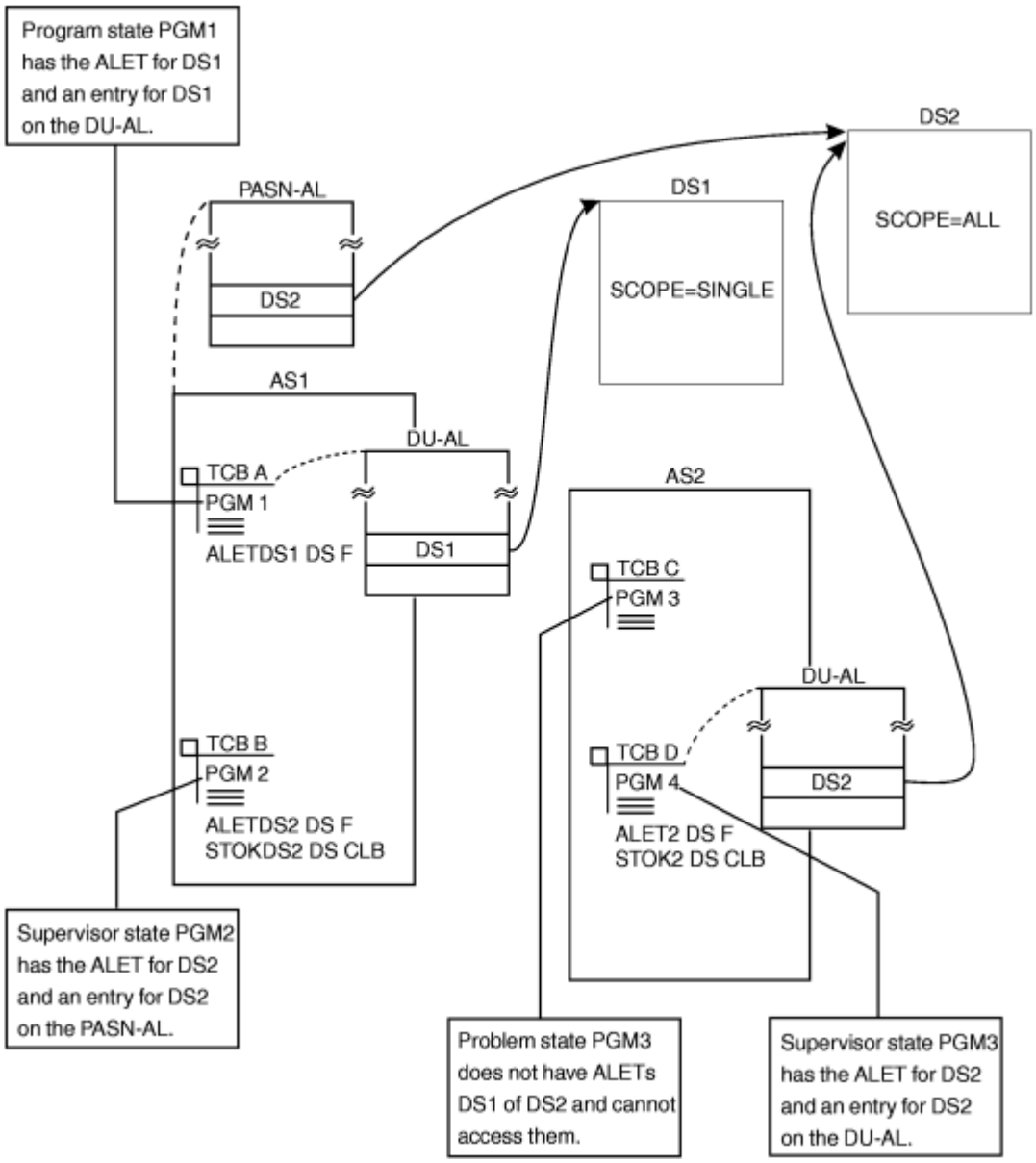


Figure 40. Example of Rules for Accessing Data Spaces

Summary of rules for creating, deleting, and using data spaces

Table 13 on page 132 summarizes the rules for what programs can do with data spaces. The third column describes what a problem state program can do if it is PSW key 8 through F. The fourth column describes what a supervisor state program or any program having PSW key 0 through 7 can do.

Table 13. Creating, Deleting, and Using Data Spaces

Function	Type of data space	A problem state, key 8 - F program:	A supervisor state or key 0-7 program:
CREATE	SCOPE=SINGLE	Can create a SCOPE=SINGLE data space.	Can create the data space if its primary or home address space is the same as the intended owner's home address space
	SCOPE=ALL SCOPE=COMMON	Cannot create the data spaces.	Can create the data space if its primary or home address space is the same as the intended owner's home address space
DELETE	SCOPE=SINGLE	Can delete the SCOPE=SINGLE data spaces it owns or created if its PSW key matches the storage key of the data space.	Can delete a SCOPE=SINGLE data space if its primary or home address space is the same as the owner's home address space.
	SCOPE=ALL SCOPE=COMMON	Cannot delete the data space.	Can delete the data space if its primary or home address space is the same as the owner's home address space.
RELEASE	SCOPE=SINGLE	Can release storage in the data spaces it owns or created if its PSW key matches the storage key of the data space.	Can release storage in a SCOPE=SINGLE data space if its primary or home address space is the same as the owner's home address space and its PSW key matches the storage key of the data space.
	SCOPE=ALL SCOPE=COMMON	Cannot release the storage.	Can release storage in the data space if its PSW key matches the storage key of the data space.
EXTEND	SCOPE=SINGLE SCOPE=ALL SCOPE=COMMON	Can extend the current size if it owns the data space.	Can extend the current size.
LOAD or OUT	SCOPE=SINGLE SCOPE=ALL SCOPE=COMMON	Can page areas into (and out of) central storage from (or to) a data space created by any task in its address space.	Can page areas into and out of central storage.
Add entries to the DU-AL	SCOPE=SINGLE	Can add entries for the SCOPE=SINGLE data spaces it owns or created.	Can add entries for a SCOPE=SINGLE data space if the caller's home and owner's home address space are the same.
	SCOPE=ALL SCOPE=COMMON	Cannot add the entries.	Can add entries for the SCOPE=ALL (not the SCOPE=COMMON) data space.

Table 13. Creating, Deleting, and Using Data Spaces (continued)			
Function	**Type of data space**	**A problem state, key 8 - F program:**	**A supervisor state or key 0-7 program:**
Add entries to the PASN-AL	SCOPE=SINGLE	Can add entries if it owns or created the data space, and the data space is not already on the PASN-AL as a result of an ALESERV ADD issued by a problem state program with PSW key 8 - F.	Can add entries for a SCOPE=SINGLE data space if its PASN-AL is the same as the PASN-AL of the owner's home address space.
	SCOPE=ALL SCOPE=COMMON	Cannot add entries.	Can add entries for a SCOPE=ALL and a SCOPE=COMMON data space.
Access a data space through a DU-AL or PASN-AL	SCOPE=SINGLE SCOPE=ALL SCOPE=COMMON	Can access a data space through an access list if the entry for the data space exists and the program has the ALET. Data space storage is subject to storage key and fetch protection.	Can access a data space through an access list if the entry for the data space exists and the program has the ALET. Data space storage is subject to storage key and fetch protection.

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. The entire data space has the storage key that you request, or, by default, the storage key that matches your own PSW key.

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 the data space” on page 134.
- 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:

- A request for a SCOPE=ALL or SCOPE=COMMON data space. If you don't code SCOPE, the system creates a SCOPE=SINGLE data space. See “Scope=single, scope=all, and scope=common data spaces” on page 128.
- 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 the data space” on page 134.
- 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 the data space” on page 136.
- A request that the system create a data space where disabled programs can access data (DREF parameter). See “Creating a data space of DREF storage” on page 137.
- A request that the data space be fetch-protected (FPROT parameter). See “Protecting data space storage” on page 137.

- The storage key of the data space (KEY parameter). Use CALLERKEY to specify that the storage key of the data space is to match your PSW key (or take the default for the KEY parameter). See "Protecting data space storage" on page 137.
- The TTOKEN of the TCB to which you assign ownership of the data space (TTOKEN parameter). See "How SRBs use data spaces" on page 151.

Choosing the name of the data space

The name you specify on the NAME parameter will identify the data space on some dump requests and IPCS commands.

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 for your data space. 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 *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

Use the GENNAME parameter to ask the system to generate a unique name for your data space. 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 you are creating, use the OUTNAME parameter.

Note: The maximum number of system-generated names is 99,999. If all system-generated names have been used, DPSERV reuses generated names from previously deleted data spaces or hiperspaces. If all system-generated names are in use for active data spaces or hiperspaces, DSPSERV fails with a return code of "08" and a reason code of "0012". Before the maximum number of system-generated names is reached, the counter will not be reset to zero until all data spaces and hiperspaces within the address space are deleted. The generated names counter will be reset to zero when the job is recycled.

Therefore, if your program is a batch job and it is creating a data space, do not:

- Request that the system generate a name (through the GENNAME parameter), ***and***
- Assign ownership to a TCB that remains for the life of the address space.

Specifying the size of the data space

When you create a data space, you tell the system on the BLOCKS parameter how large to make that space, the largest size being 524,288 blocks. (The product of 524,288 times 4K bytes is 2 gigabytes.) The addressing range for the data space 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 that have a storage key of 8 through F. If your request for a data space 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.

If you create the data space with a storage key of 0 through 7, the system does not check the size against the total storage already used for data spaces and hiperspaces. If you create the data space with a storage key of 8 through F, the system adds the initial size of the space to the cumulative total of all data spaces and hiperspaces for the address space and checks this total against the installation limit for an address space.

For information on the IBM defaults and how to change them, see "Limiting data space use" on page 139.

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, you can use the DSPSERV EXTEND macro to increase the size of the available storage, thus increasing the storage in the data space that is available for the program. The amount of available storage is called the **current size**. (At the creation of a data space, 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 of the data space.

If you know the default size and want a data space smaller than or equal to that size, use the BLOCKS=maximum size or omit the BLOCKS parameter.

If you know what size data space 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 data space with the maximum size and the initial size both set to the default size.

If you do not know how large a data space (with storage key 8 - F) 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 data 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 41 on page 136 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 of 20,000 bytes.

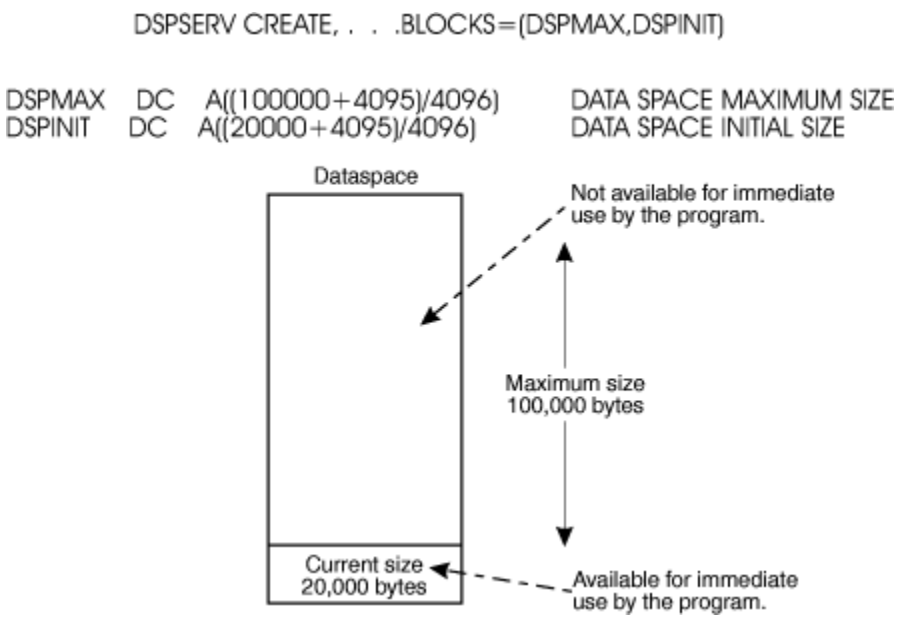


Figure 41. Example of Specifying the Size of a Data Space

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 143 describes extending the current size and includes an example of how to extend the current size of the data space in Figure 41 on page 136.

Identifying the origin of the data space

Some processors do not allow the data space to start at zero; these data spaces start at address 4096 bytes. When you use DSPSERV CREATE, you can count on the origin of the data space staying the same within the same IPL. To learn the starting address, either (1) create a data space of 1 block of storage more than you need and then assume that the data space starts at 4096 or (2) use the ORIGIN parameter. If you use ORIGIN, the system returns the beginning address of the data space 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 described as follows:

> Suppose a program creates a data space of 1 megabyte and assumes the data space starts at address zero 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
```

The data space that the system creates has the same storage protection key as the PSW key of the caller.

Protecting data space storage

If the creating program wants the data space to have read-only access, it can use the FPROT and KEY parameters on DSPSERV. KEY assigns the storage key for the data space and FPROT specifies whether the storage in the data space is to be fetch-protected. Storage protection and fetch protection rules apply for the entire data space. For example, a program cannot reference storage in a fetch-protected data space without holding the PSW key that matches the storage key of the data space or PSW key 0.

Figure 42 on page 137 shows a SCOPE=ALL data space DSX with a storage key of 5, owned by a subsystem. PGM1 and PGM2, with PSW keys of 8, have entries for the data space on their DU-ALs and have the ALETs for these entries. However, their PSW keys do not match the storage key of the data space. Their ability to access data in DSX depends on how the creating program coded the FPROT parameter on the DSPSERV macro.

- If the creating program specified no fetch-protection (FPROT=NO), PGM1 and PGM2 can fetch from but not store into the data space.
- If the creating program specified fetch-protection (FPROT=YES), PGM1 and PGM2 can neither fetch from nor store into the data space.

Figure 42 on page 137 shows one way PGM1 and PGM2 can gain fetch and store capability to the data space. The subsystem provides a PC routine with a PSW key of 5 in the common area. To access the data space, the two users PC into the subsystem's address space and have access to its data space.

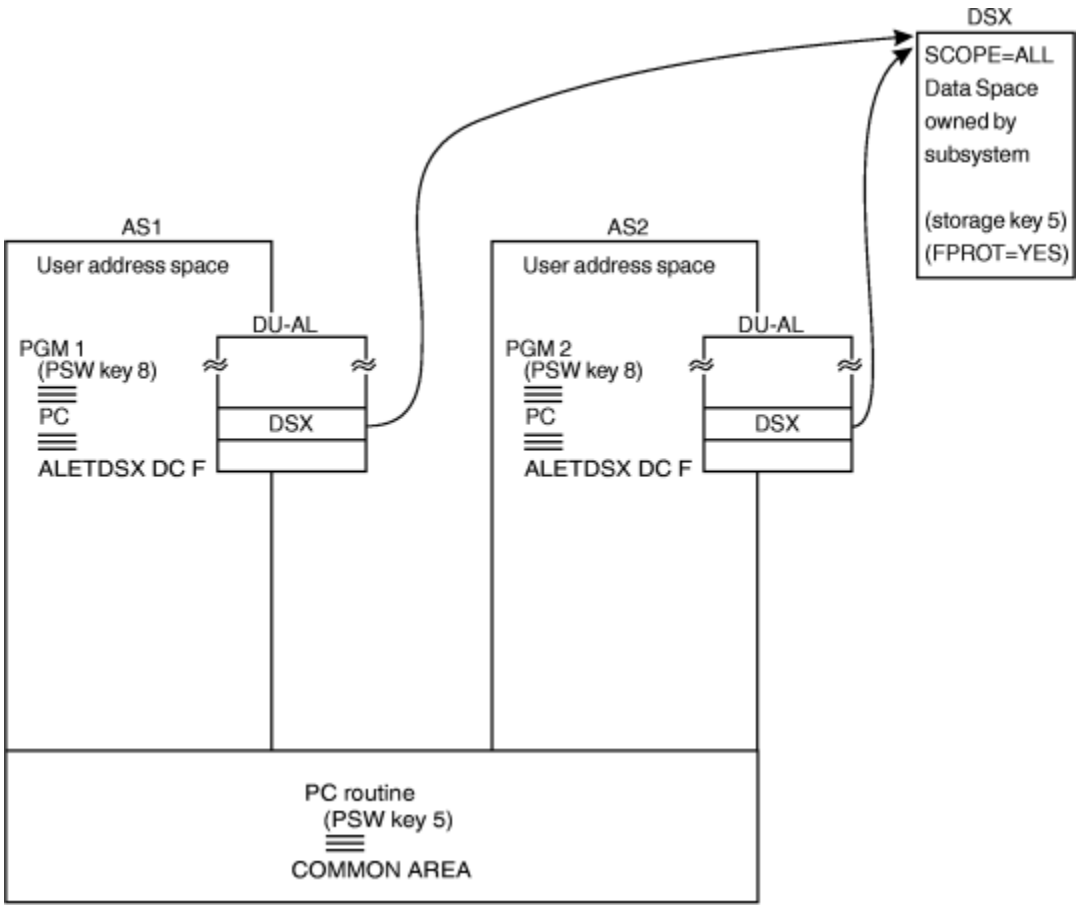


Figure 42. Protecting storage in a data space

Creating a data space of DREF storage

Through the DSPSERV macro, supervisor state and PSW key 0 - 7 programs can create a data space that consists of disabled reference (DREF) storage. **DREF storage** is storage that can be referenced by callers that are running disabled. DREF storage uses more system resources than non-DREF storage because the system does not page DREF storage out to auxiliary storage. Instead, it uses central storage (and

expanded storage, if your processor has it). **IBM recommends** that you not use DREF storage when pageable storage is sufficient.

To request DREF storage, code DREF=YES on the DSPSERV CREATE macro. A data space with DREF storage can be SCOPE=SINGLE, SCOPE=ALL, or SCOPE=COMMON.

Establishing addressability to a data space

Creating a data space does not give you addressability to that data space. Before you can use the data space, you must issue the ALESERV macro, which adds an entry to an access list and returns the ALET that indexes the entry. Examples of this process appear in this chapter; Chapter 5, “Using access registers,” on page 93, contains additional examples.

When you use ALESERV, you can omit the ACCESS parameter, which specifies whether an access list entry is *public* or *private*. Data space entries are always public, the default for ACCESS.

Example of establishing addressability to a data space

In the following example, a program establishes addressability to a data space named TEMP. Input to the ALESERV macro is the STOKEN that the DSPSERV macro returned. ALESERV places an entry on the DU-AL and returns the ALET for the data space.

```
         ALESERV ADD,STOKEN=DSPCSTKN,ALET=DSPCALET,AL=WORKUNIT
         .
DSPCSTKN DS   CL8                     DATA SPACE STOKEN
DSPCALET DS   F                       DATA SPACE ALET
```

Managing data space storage

Managing storage in data spaces differs from managing storage in address spaces. Keep the following advisory notes in mind:

- When you create a data space, request a maximum size large enough to handle your application's needs and, optionally, an initial size large enough to meet its immediate needs.
- You can use callable cell pool services to keep track of data space storage. (The STORAGE, GETMAIN, FREEMAIN, or CPOOL macros do not manage data space storage.) For information about how to use callable cell pool services and an example of its use with data spaces, see “Using callable cell pool services to manage data space areas” on page 141.
- If you are not going to use an area of a data space again, release that area to free the resources that back the area.
- When you are finished using a data space, remove its entry from the access list and delete the data space.

Managing data space storage across a checkpoint/restart operation

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 perform a DSPSERV CREATE to recreate the data space.
  2. Perform an 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 perform an ALESERV ADD to access the data space.

See *z/OS DFSMSdfp Checkpoint/Restart* for more information about the CHKPT macro.

Limiting data space use

The use of data spaces consumes system resources such as virtual, processor, and auxiliary storage. Programmers responsible for tuning and maintaining MVS can control the use of these resources. Through the system management facility (SMF) installation exit IEFUSI, an installation can set limits on the amount of virtual storage that programs in each address space can use for data spaces and hiperspaces.

For information on using IEFUSI, see *z/OS MVS Installation Exits*.

Serializing use of data space storage

At many installations, users must share access to data in a data space. Users who are updating data for common use by other programs need exclusive access to that data during the updating operation. If several users tried to update the same data at the same time, the result would be incorrect or damaged data. To protect the integrity of the data, you might need to serialize access to the data in the data space.

Serializing the use of the storage in a data space requires methods like those you would use to serialize the use of virtual storage in an address space. Use the ENQ and DEQ macros, compare and swap operations, or establish your own protocol for serializing data space use.

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) or Move Long (MVCL) instruction 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 or within a data 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:

- When you are in AR mode, do not use AR/GPR 0 in the MVCL instruction. In Example 2, the MVCL instruction uses GPRs 2, 3, 4, and 5.
- The maximum amount of data that one execution of the MVCL instruction can move is 16,777,215 bytes.

Using callable cell pool services to manage data space areas

You can use the callable cell pool services to manage the virtual storage of 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 three different areas:

- One anchor
- Up to 65,000 extents
- 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 the section on callable cell pool services in *z/OS MVS Programming: Assembler Services Guide*.

Example of Using Callable Cell Pool Services with a Data Space

Assume that you have an application that requires up to 4,000 records that are each 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 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. In 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 one byte for every eight cells. In this example, adding 128 to 500 (that is, 4000 divided by 8) equals 628 bytes. The system then rounds up to a doubleword making the extent 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 (632) 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 43 on page 142 describes the areas you have defined in the data space.

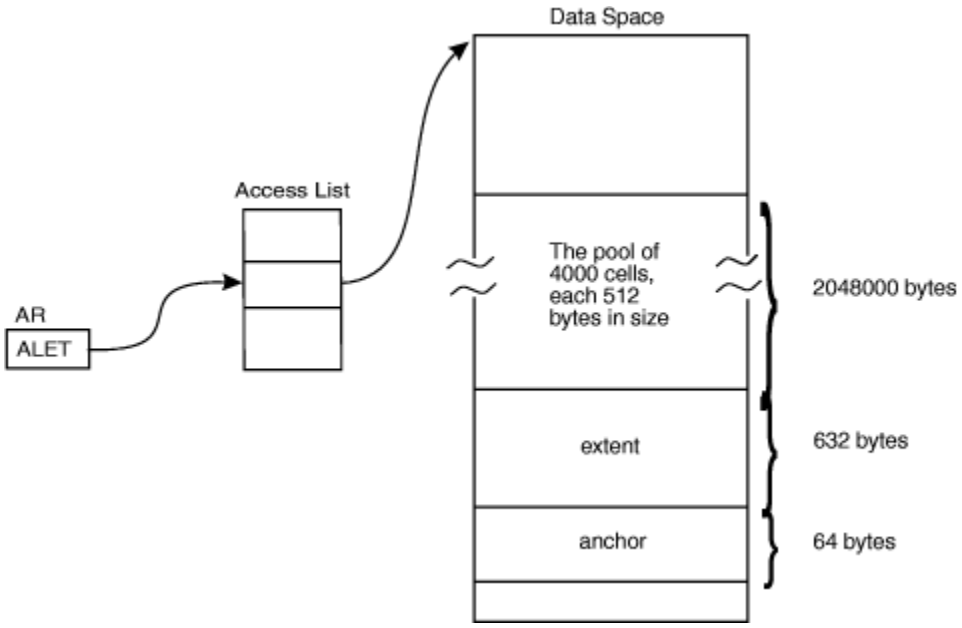


Figure 43. Example of Using Callable Cell Pool Services for Data Spaces

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.

Extending the current size of a data space

When you create a data space and specify an initial size smaller than the maximum 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
  - Reject the request, if the data space has storage key 8 - F and the request would exceed the installation limits

Consider the data space in Figure 41 on page 136, where the current (and initial) size is 20,000 bytes and the maximum size is 100,000 bytes. To increase the current size to 50,000 bytes, adding 30,000 bytes to the current size, the creating program would code the following:

```
          DSPSERV EXTEND,STOKEN=DSSTOK,BLOCKS=DSBLCKS
          .
DSDELTA EQU  30000                    30000 BYTES OF SPACE
DSBLCKS DC   A((DSDELTA+4095)/4096)   NUMBER OF BLOCKS ADDED TO DATA SPACE
DSSTOK  DS   CL8                      STOKEN RETURNED FROM DSPSERV CREATE
```

The storage the program can use would then be 50,000 bytes, as shown in Figure 44 on page 143.

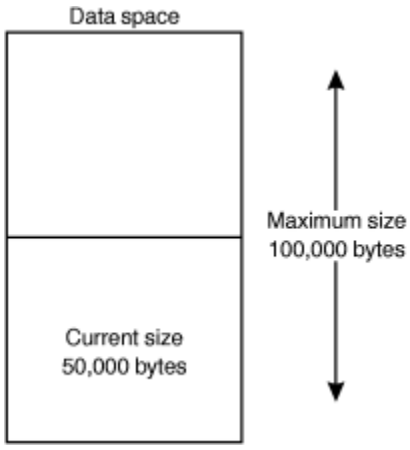


Figure 44. Example of Extending the Current Size of a Data Space

If you use VAR=YES when you issue the EXTEND request, use NUMBLKS to find out the size by which the system extended the data space.

Deleting a data space

When a task doesn't need the data space any more, it can free the virtual storage and remove the entry from the access list.

A problem program with PSW key 8 - F can delete only the data spaces it created or owns, provided it has a PSW key that matches the storage key of the data space.

Example of deleting a data space

The following example shows you how to delete a data space entry from an access list and then delete the data space.

```
         ALESERV DELETE,ALET=DSPCALET       REMOVE DS FROM AL
         DSPSERV DELETE,STOKEN=DSPCSTKN     DELETE THE DS
         .
DSPCALET DS   F                             DATA SPACE ALET
DSPCSTKN DS   CL8                           DATA SPACE STOKEN
```

IBM recommends that you explicitly remove the entry for a data space from the access list and delete the space before the owning task terminates. This frees up resources when they are no longer needed, and avoids excess processing at termination time. However, if you don't, MVS automatically does it for you at termination time.

Example of creating, using, and deleting a data space

This section contains a complete example of a how a problem program creates, establishes addressability to, uses, and deletes 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. However, you should always check the return codes after issuing the macro.

The lines of code in the middle of the example (under the comment "MANIPULATE DATA IN THE DATA SPACE") 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         SET GLOBAL BIT FOR AR MODE
         .
*  ESTABLISH AR/GPR 12 AS BASE REGISTER
         .
         LAE   12,0                 SET BASE REGISTER AR
         BASR  12,0                 SET BASE REGISTER GPR
         USING *,12
```

```
*  CREATE THE DATA SPACE AND ADD THE ENTRY TO THE ACCESS LIST
         .
         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 GR2
         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
         .
```

```
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                         DATA SPACE STORAGE MAPPING
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.

Creating and using SCOPE=COMMON data spaces

The SCOPE=COMMON data space provides your programs with virtual storage that is addressable from all address spaces and all programs. In many ways, it is the same as the common service area (CSA) of an address space. You might use a SCOPE=COMMON data space instead of CSA because:

- A SCOPE=COMMON data space offers up to two gigabytes of commonly addressable virtual storage for data (but not executable code). The CSA offers a much smaller amount of storage.
- The CSA is a limited resource; because it is a part of all address spaces, the use of this virtual storage area reduces the amount of common area available for all programs.

To create this space, use the SCOPE=COMMON parameter on DSPSERV CREATE. You can use any of the parameters on that macro to establish the characteristics of that space.

To gain addressability to the space, issue the ALESERV ADD macro with the AL=PASN parameter. ALESERV ADD then adds an entry for the data space to the caller's PASN-AL and returns the ALET for that entry. Additionally, ALESERV ADD adds the same entry to every PASN-AL in the system. As new address spaces come into the system, their PASN-ALs have this entry on them. **All programs use the same ALET to access the data space.** In other words, with the entry on all PASN-ALs, programs in other address spaces do not have to issue the ALESERV ADD macro. However, the creating program must pass the ALET for the data space to the other programs.

The use of the virtual storage in the SCOPE=COMMON data space is similar to the use of the CSA. A program wanting to share CSA storage with another program has to pass the address of that area to the other program; the creator of the SCOPE=COMMON data space has to pass the ALET value to the other program. (It might also have to tell the other program the origin of the data space.)

Figure 45 on page 146 shows an example of a SCOPE=COMMON data space named COMDS that PROG1 created. PROG1 uses ALESERV ADD to add an entry to its PASN-AL. Because COMDS is

SCOPE=COMMON,that same entry appears on all PASN-ALs in the system, plus all PASN-ALs that will exist from the time the entry for the SCOPE=COMMON data space is added to the access list until the data space terminates. PROG1 has the ALET for the entry. To give access to COMDS to programs in the other address spaces, PROG1 passes the ALET to the other programs.

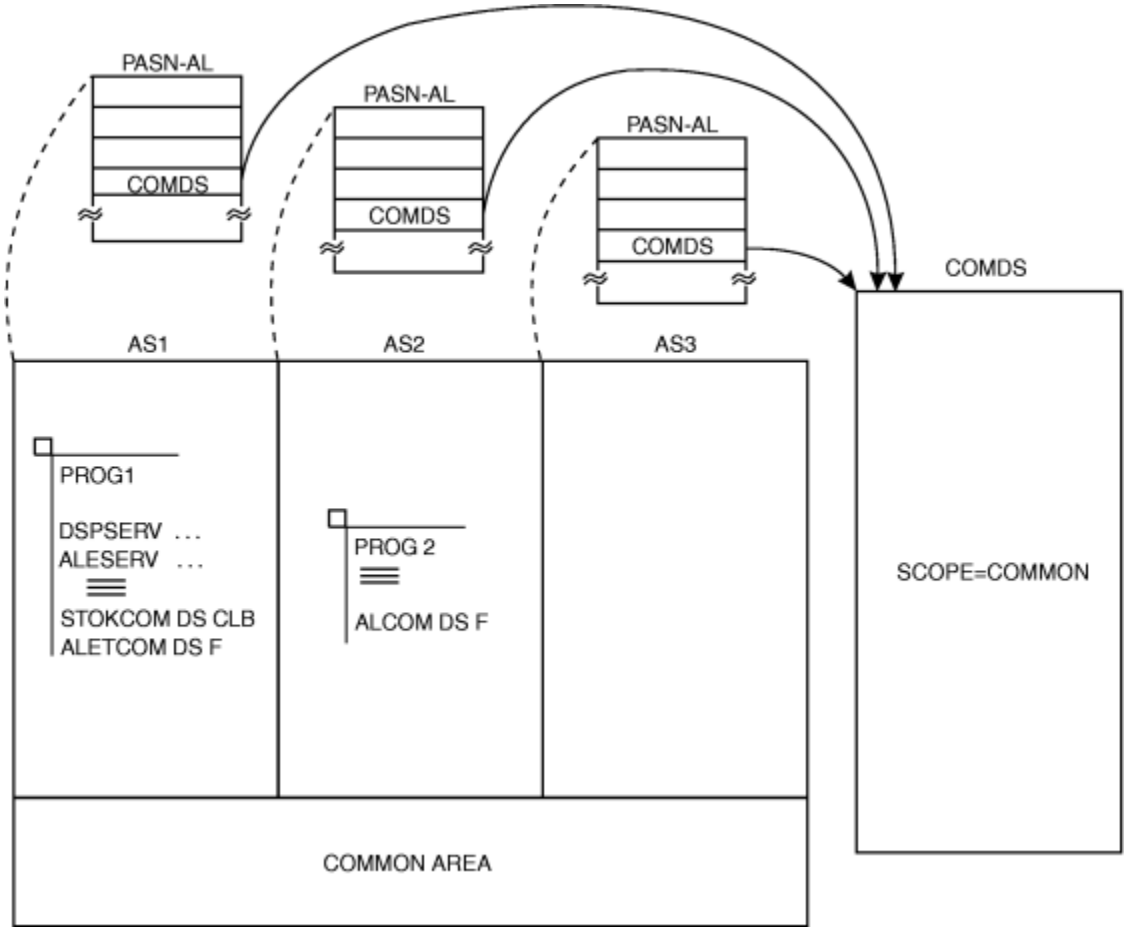


Figure 45. Example of Using a SCOPE=COMMON Data Space

Programming Considerations: When you use SCOPE=COMMON data spaces, keep in mind the following advice:

- Use the SCOPE=COMMON data space when your program has large amounts of data that it wants to share across multiple address spaces. For example, to share more than 10 megabytes of commonly addressable data, consider using a SCOPE=COMMON data space. To use less than 10 megabytes, consider using CSA.
- To make sure problem state programs cannot access the SCOPE=COMMON data space, use the FPROT and KEY parameters to assign fetch protection and a specific storage key.

  For example, consider that PROG1 in Figure 45 on page 146 used the following parameters on DSPSERV when it created COMDS:

  ```
  FPROT=YES,KEY=5
  ```

  In this case, only programs with PSW key 5 or PSW key 0 can access the data in COMDS. A TSO/E user (with PSW key 8) would then be unable to either store into or fetch from the data space.
- SCOPE=COMMON data spaces can only be assigned a system storage key (0-7).
- The system can reuse the ALET associated with a SCOPE=COMMON data space after the space terminates. Therefore, manage the termination and reuse of ALETs for the SCOPE=COMMON data space. This action is described in “ALET reuse by the system” on page 114.

- To help solve system problems and error conditions, use the data space dumping services to dump appropriate areas of the SCOPE=COMMON data space. See "Dumping storage in a data space" on page 156 for information about dumping data space areas.

Your installation can use the IEASYSxx member of SYS1.PARMLIB to set limits on the total number of SCOPE=COMMON data spaces available to programs. For information about how to set up this member, see *z/OS MVS Initialization and Tuning Reference*.

Attaching a subtask and sharing data spaces with it

A program, whether in supervisor state or problem state, can use the ALCOPY=YES parameter on the ATTACH or ATTACHX macro to attach a subtask and pass a copy of its DU-AL to this subtask. In this way, the program can share data spaces or hiperspaces with a program running under the subtask. The two programs both have access to the address/data spaces and hiperspaces that have DU-AL entries at the time of the ATTACH or ATTACHX macro invocation. Note that it is not possible to pass only a part of the DU-AL.

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.

The following example, represented by Figure 46 on page 147, assumes that program PGM1 (running under TCBA) has created a SCOPE=SINGLE data space DS1 and established addressability to it. Its DU-AL has several entries on it, including one for DS1. PGM1 uses the ATTACHX macro to attach subtask TCBB. PGM1 uses the ALCOPY=YES parameter to pass a copy of its DU-AL to TCBB. It can also pass ALETs in a parameter to PGM2. Upon return from ATTACHX, PGM1 and PGM2 have access to the same data/address spaces.

The figure shows the two programs, PGM1 and PGM2, sharing the same data space.

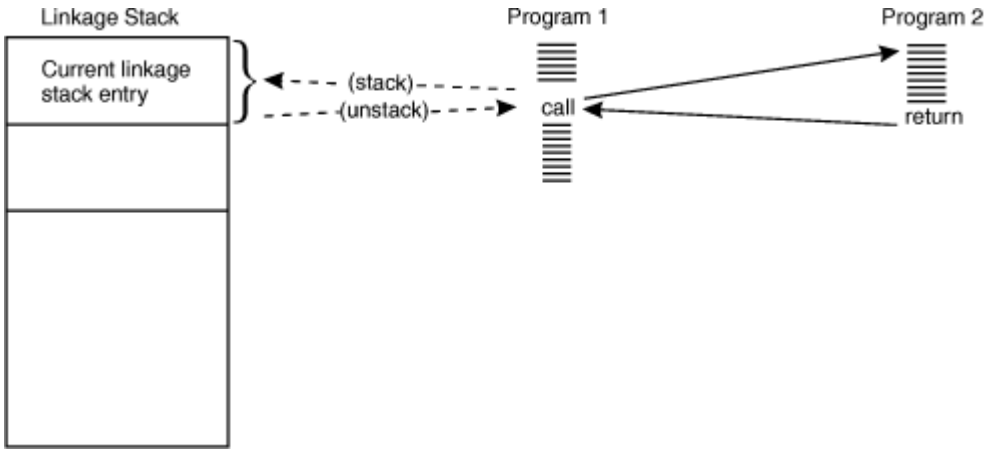


Figure 46. Two programs sharing a SCOPE=SINGLE data space

Example of attaching a task and passing a DU-AL

The following example shows you how TCBA attaches TCBB and passes its DU-AL:

```
   DSPSERV CREATE,NAME=DSNAME,BLOCKS=DSSIZE,STOKEN=DSSTOK,ORIGIN=DSORG
   ALESERV ADD,STOKEN=DSSTOK,ALET=DSALET
   ATTACHX EP=PGM2,ALCOPY=YES
    .
DSNAME      DC   CL8'MYDSPACE'   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 two DU-ALs do not necessarily stay identical; after the attach, PGM1 and PGM2 are free to add and delete entries on their own DU-ALs.

If TCBA terminates, the system deletes the data space that belonged to TCBA and terminates PGM2.

Sharing data spaces among problem state programs with PSW key 8 through F

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 4K UNITS
JSTTTOK     DS  CL8             TTOKEN OF JOB STEP TASK
```

Unless PGM1 or 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.

Mapping a data-in-virtual object to a data space

Through data-in-virtual, your program can map a data-in-virtual object to a data space. The data-in-virtual object must be a VSAM linear data set. Use DIV macros to set up the relationship between the object and the data space. Setting up the relationship between the object and the data space 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 the data space.

Supervisor state programs or problem state programs with PSW key 0 - 7 (called "authorized programs" in this section) can issue DIV IDENTIFY and then have subtasks of that task use the DIV services (except the ACCESS service) for the same ID. The subtasks must also be authorized. This means that an authorized program can issue a DIV IDENTIFY and an authorized subtask can issue the DIV MAP for that ID.

Table 14 on page 149 shows what data-in-virtual requires of the tasks that represent the authorized programs that issue the DIV macros. The table does not show the IDENTIFY service because data-in-virtual does not have restrictions on this service.

Table 14. Requirements for authorized programs using the DIV services with data spaces

	ACCESS	**MAP**	**SAVE**	**UNIDENTIFY, UNACCESS, UNMAP, RESET**
Object is a linear data set, window is in a data space	Task that issued the DIV IDENTIFY.	Task (or authorized subtask of the task) that issued the DIV IDENTIFY. (See Notes.)	Task (or authorized subtask of the task) that issued the DIV IDENTIFY. The task does not have to own the data space.	Task (or authorized subtask of the task) that issued the DIV IDENTIFY. The task does not have to own the data space.

- If the program is in supervisor state or PSW key 0 - 7, any task within the caller's primary address space can own the data space.
- If the program is APF-authorized, but not supervisor state or PSW key 0 - 7, the caller must own or be the creator of 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.

Example of mapping a data-in-virtual object to a data space

Figure 47 on page 149 shows a data-in-virtual object mapped into a data space. The "window" is the entire data space.

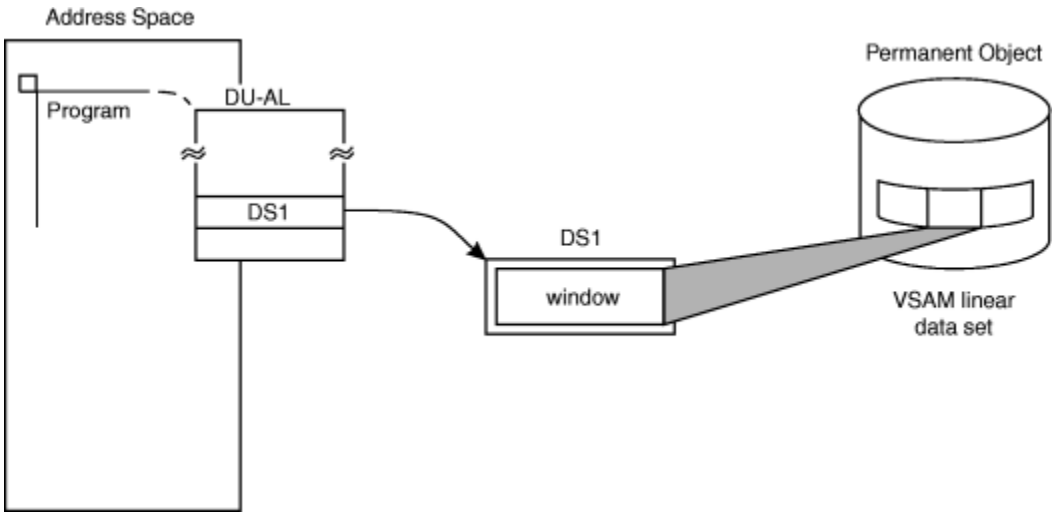


Figure 47. Example of Mapping a Data-in-Virtual Object to a Data Space

The following example maps a data-in-virtual object into the data space illustrated in Figure 47 on page 149. 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=DSORG,STOKEN=DSSTOK
  .
* USE THE ALET IN DSALET TO REFERENCE THE
* DATA SPACE STORAGE MAPPING THE OBJECT.
  .
  MVC     OBJWORD1,DATAIN
  MVC     OBJWORD2,DATA2
  .
```

```
* 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
DATAIN   DC   CL4'JOBS'
DATA2    DC   CL4'PAYR'
OBJAREA  DSECT                     WINDOW IN DATA SPACE
OBJWORD1 DS   F
OBJWORD2 DS   F
```

See the section on data-in-virtual in *z/OS MVS Programming: Assembler Services Guide* for more help in using data spaces with data-in-virtual.

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 does not have to be on a 4K-byte boundary, nor does the size have to be an increment of 4K blocks. (Note that DSPSERV LOAD performs the same action for a data space 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 block of data space storage, you can use DSPSERV OUT to page the area out of central storage. The system will make these real storage frames available for reuse. DSPSERV OUT requires that you specify the STOKEN, the beginning address of the area, and the size of the area. (Note that DSPSERV OUT corresponds to the PGSER macro with the OUT parameter.)

When your program has no further need for the data in a certain area of a data space, it can use DSPSERV RELEASE to free that storage.

Releasing data space storage

Your program can 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 (that is, initialize to hexadecimal zeroes and return the resources to the system) the virtual storage of a data space, use the DSPSERV RELEASE macro. 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.

To release storage in a data space, the caller must have a PSW key that is either zero or equal to the key of the data space storage the system is to release. If the caller is in supervisor state with PSW key 0 - 7 and is releasing a SCOPE=SINGLE data space, the caller's home or primary address space must be the same as the owner's home address space. If the caller is in problem state with PSW key 8 - F and is releasing a SCOPE=SINGLE data space, the caller must own or have created the data space. Otherwise, the system abnormally ends the caller.

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 processor or auxiliary storage.

If your program is running disabled for I/O or external interrupts, use the DISABLED=YES parameter on DSPSERV RELEASE. If your program is running disabled and issues DSPSERV RELEASE without DISABLED=YES, the system abends the program.

How SRBs use data spaces

An SRB cannot own a data space. Through the DSPSERV CREATE macro, a supervisor state or PSW key 0-7 program running under an SRB must assign ownership of a data space to a TCB. The owning TCB must reside in the SRB's home or primary address space.

Like a TCB, an SRB routine has a DU-AL and can use the PASN-AL of its address space. The DU-AL that the system gives the SRB routine can be either empty or a copy of the scheduling program's DU-AL. When you issue the SCHEDULE macro to schedule an SRB, you can obtain:

- An empty DU-AL for the SRB routine by specifying MODE=NONXM. With a mode of NONXM, the SRB routine runs with its primary, secondary, and home address spaces equal to SRBASCB.
- A copy of the scheduling routine's DU-AL by specifying MODE=FULLXM. If the scheduling program creates entries in the DU-AL after scheduling the SRB, the SRB routine will not have access to those data spaces. With a mode of FULLXM, the SRB runs with the same primary, secondary, and home addressability as the scheduling program.

Figure 48 on page 152 and Figure 49 on page 153 illustrate the attributes of an SRB that is scheduled with MODE=FULLXM and MODE=NONXM. Table 15 on page 153 identifies the home, primary, and secondary addressability for each type of invocation of the SCHEDULE macro.

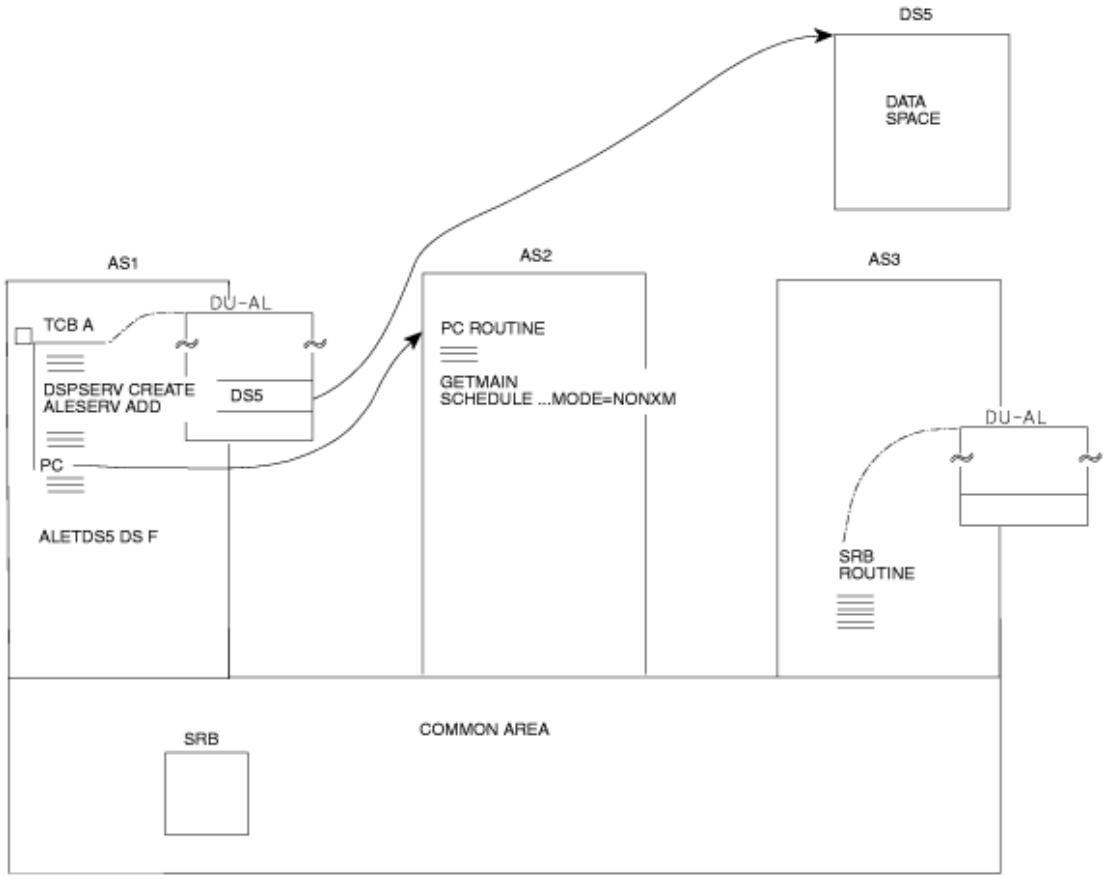


Figure 48. Scheduling an SRB with an empty DU-AL and in a non-cross memory environment

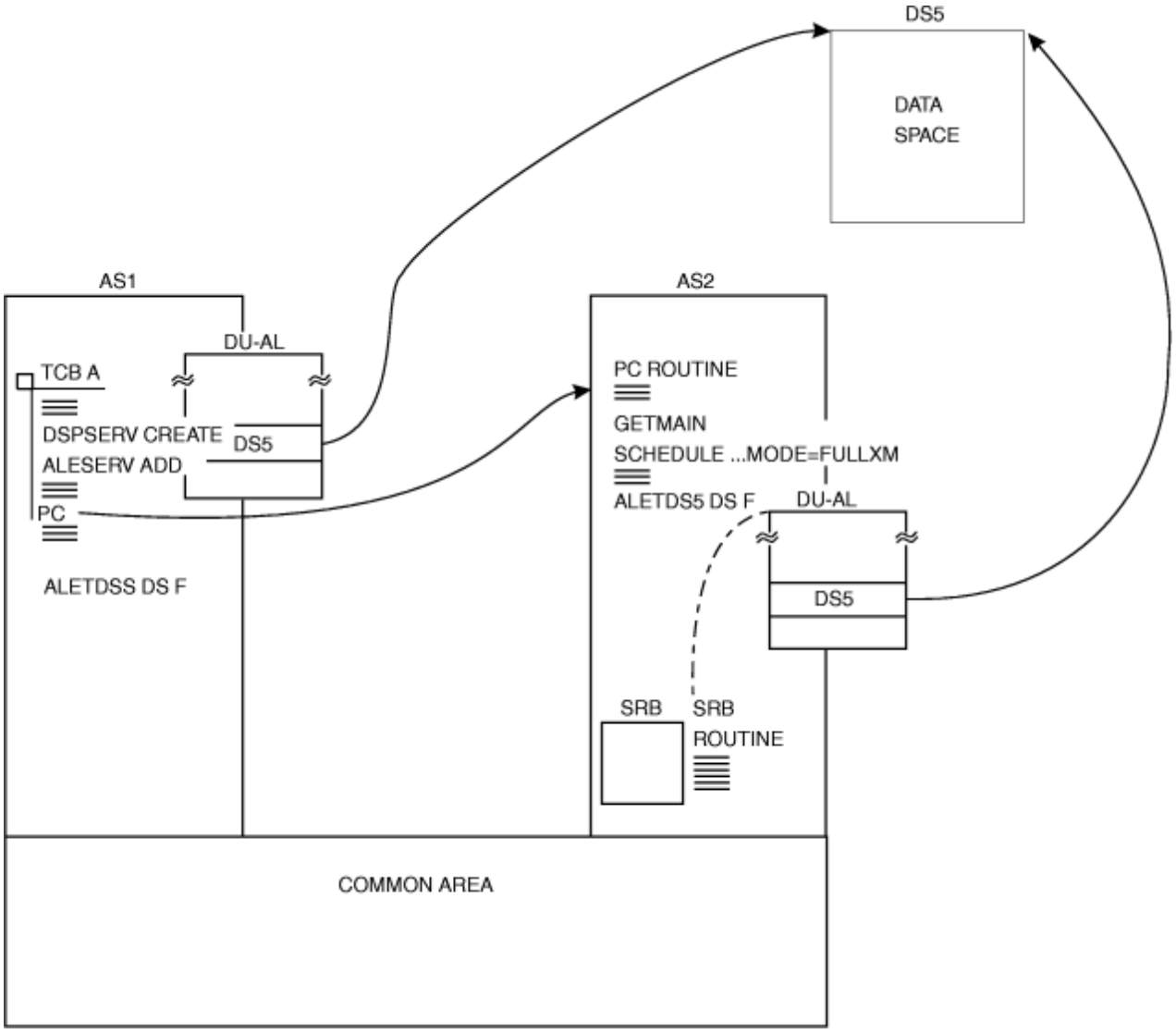


Figure 49. Scheduling an SRB with a copy of the scheduling program's DU-AL and in the same cross memory environment

Table 15. Addressability for each type of invocation of the SCHEDULE macro

	NONXM		**FULLXM**	
	TCB	**SRB**	**TCB**	**SRB**
HOME	AS1	AS3	AS1	AS1
Primary	AS1	AS3	AS1	AS1
Secondary	AS2	AS3	AS2	AS2

When you use the DSPSERV CREATE macro to create the data space and assign ownership, you must identify the TCB through the TTOKEN parameter. A TTOKEN identifies a TCB. Unlike TCB addresses, TTOKENs are unique within the IPL; the system does not assign this same identifier to any other TCB until the next IPL. If you know the TCB address of the task that is to receive ownership, but not the TTOKEN, use the TCBTOKEN macro. The TCBTOKEN macro accepts the TCB address and returns a TTOKEN. You then use this TTOKEN in the DSPSERV CREATE macro.

For more information about TTOKENs, see “Obtaining the TCB identifier for a task (ttoken)” on page 154.

When an SRB routine terminates, it can delete any data spaces it created. Use the STOKEN parameter on the DSPSERV DELETE macro to specify the data space.

Obtaining the TCB identifier for a task (ttoken)

Each task in the system is identified in two ways:

- By the TCB address.
- By the TTOKEN of the task. A TTOKEN is an identifier that the system assigns to a TCB. Unlike a TCB address, a TTOKEN is unique within the IPL; the system does not assign the same identifier to any other TCB until the next IPL.

Some MVS macros require that you identify the task using the TCB address, some require the TTOKEN, and some allow you to use either the TCB address or the TTOKEN. If you know a task's TCB address and need the TTOKEN value or if you need the TTOKEN for the current task, the task that attached the current task, or the job step task, you can use the TCBTOKEN macro to obtain the value. You can also use the TCBTOKEN macro if you know the TTOKEN for a task and want the TCB address. Use the TYPE parameter on the TCBTOKEN macro to specify the value you are looking for:

TOTTOKEN
The system returns the TTOKEN of the task whose TCB address you specify.

CURRENT
The system returns the TTOKEN of the currently active task.

PARENT
The system returns the TTOKEN of the task that attached the currently active task.

JOBSTEP
The system returns the TTOKEN of the job step task for the primary address space.

TOTCB
The system returns the TCB address for the task whose TTOKEN you specify.

Example of an srb routine using a data space

In the following example, an SRB routine creates a data space, assigning ownership to the scheduling TCB (that is, the TCB that represents the program that schedules the SRB). The example includes the deletion of the data space. To assign the ownership, the routine must know the TTOKEN of the TCB. For this example, assume that the scheduling program has passed the address of the scheduling TCB through the user field in the SRB, SRBPARM. (The system loads the address of this field into GPR 1 when the system dispatches the SRB.) Before it creates the data space, the routine uses the scheduling TCB address as input to the TCBTOKEN macro to obtain the TTOKEN of the TCB.

```
*  EXAMPLE ASSUMES CALLER RUNNING WITH PASN=HASN AND THE
*    DATA SPACE WILL BE OWNED BY THE TCB THAT SCHEDULED THE SRB
         .
         BAKR  14,0                    SAVE CALLER'S STATUS ON STACK
         LAE   10,0                    SET BASE REGISTER AR
         BASR  10,0                    SET BASE REGISTER GR
         USING *,10
         SAC   0                       ENSURE IN PRIMARY MODE
         SYSSTATE ASCENV=P             SET THE GLOBAL BIT
         .
*  GET HOME ADDRESS SPACE LOCAL LOCK FOR THE TCBTOKEN SERVICE
         .
         SETLOCK OBTAIN,TYPE=LOCAL,MODE=UNCOND,REGS=USE
         .
         USING PSA,0
         L     2,PSAAOLD               GET HOME ASCB ADDRESS
         .
*  GET ADDRESS OF SCHEDULING TCB (CONTENTS OF SRBPARM) FROM REGISTER 1
         .
         LR    3,1                     GET ADDRESS OF SCHEDULING TCB
         .
         TCBTOKEN TYPE=TOTTOKEN,TTOKEN=TCBTTOKN,ASCB=(2),TCB=(3)
         .
```

```
*            RELEASE LOCAL LOCK
         SETLOCK RELEASE,TYPE=LOCAL,REGS=USE
         .
         DSPSERV CREATE,NAME=DSPCNAME,STOKEN=DSPCSTKN,BLOCKS=DSPBLCKS,    X
               ORIGIN=DSPCORG,SCOPE=ALL,TTOKEN=TCBTTOKN
         .
         ALESERV ADD,STOKEN=DSPCSTKN,ALET=DSPCALET,AL=WORKUNIT
         .
*  ESTABLISH ADDRESSABILITY TO THE DATA SPACE
         .
         SAC   512                     SWITCH INTO AR ADDRESSING MODE
         SYSSTATE ASCENV=AR            SET GLOBAL BIT FOR AR MODE
         .
*  USE DATA SPACE
         .
*  DELETE DATA SPACE
         .
         ALESERV DELETE,ALET=DSPCALET     REMOVE DS FROM AL
         DSPSERV DELETE,STOKEN=DSPCSTKN   DELETE THE DS
         .
         PR                            RETURN TO CALLER
         .
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
TCBTTOKN DS   CL16                     16 BYTE FIELD FOR TCBTOKEN
```

In the following example, a TCB routine creates a data space and then schedules an SRB which can immediately address the data space.

```
*   THE SCHEDULING ROUTINE
         .
         DSPSERV CREATE,NAME=DSPCNAME,STOKEN=DSPCSTKN,BLOCKS=DSPBLCKS,    X
               ORIGIN=DSPCORG,SCOPE=ALL,TTOKEN=TCBTTOKN
         .
         ALESERV ADD,STOKEN=DSPCSTKN,ALET=DSPCALET,AL=WORKUNIT
         .
*  ESTABLISH ADDRESSABILITY TO THE DATA SPACE
         .
         SAC   512                     SWITCH INTO AR ADDRESSING MODE
         SYSSTATE ASCENV=AR            SET GLOBAL BIT FOR AR MODE
         .
*  INITIALIZE DATA SPACE
         .
*  USE DATA SPACE
         .
*  GET INTO PRIMARY ADDRESSING MODE TO ISSUE GETMAIN AND SCHEDULE
         .
         SAC   0                       SWITCH INTO PRIMARY ADDRESSING
*                                      MODE
         SYSSTATE ASCENV=PRIMARY       SET GLOBAL BIT FOR PRIMARY MODE
         .
*  OBTAIN AND INITIALIZE AN SRB AND AN SRB PARAMETER AREA
         .
         GETMAIN RU,SP=213,LV=PSRBSIZE GET THE STORAGE
         .
         USING  SRB,1
         XC     SRB,SRB                CLEAR THE SRB
         MVC    SRBPTCB,PSATOLD        SET PURGE TCB ADDRESS TO CURRENT
*                                      TCB ADDRESS
         L      8,PSAAOLD              LOCATE CURRENT ASCB
         USING  ASCB,8
         MVC    SRBPASID,ASCBASID      SET PURGE ASID TO CURRENT ASID
         MVC    SRBRMTR,RMTRADDR       SET RMTR ADDRESS
         OI     SRBRMTR,X'80000000'    SET ADDRESS TO 31-BIT MODE
         LA     7,ENTSRB               GET ENTRY POINT ADDRESS
         ST     7,SRBEP                SET ENTRY POINT ADDRESS
         OI     SRBEP,X'80000000'      SET ADDRESS TO 31-BIT MODE
         LA     2,SRBEND               PARAMETERS FOLLOW SRB
         ST     2,SRBPARM              SET PARAMETER ADDRESS
         USING  PARMS,2
         MVC    DALET,DSPCALET         SAVE DATASPACE ALET IN PARAMETERS
         XC     ECB1,ECB1              CLEAR THE ECB
         DROP   2
```

```
         .
*  SCHEDULE SRB WHICH USES THE DATASPACE
         .
         SCHEDULE SRB=(1),MODE=FULLXM
         .
*  FREE ONLY THE SRB STORAGE
         .
         FREEMAIN RU,LV=SRBSIZE,SP=213
         .
*  WAIT FOR SRB TO COMPLETE
         .
         WAIT  ECB=ECB1
         .
*  FREE THE PARAMETER STORAGE
         .
         FREEMAIN RU,LV=8,SP=213
         .
*  DELETE DATA SPACE
         .
         ALESERV DELETE,ALET=DSPCALET     REMOVE DS FROM AL
         DSPSERV DELETE,STOKEN=DSPCSTKN   DELETE THE DS
         .
         .
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
PSRBSIZE DC   A(SRBSIZE+8)             SIZE OF AN SRB PLUS AN 8 BYTE
*                                      PARAMETER AREA
RMTRADDR DC   A(RMTRXX)
*
PARMS    DSECT                         SRB PARAMETER AREA
DALET    DS   F                        DATASPACE ALET FOR SRB
ECB1     DS   F                        ECB FOR SRB TO POST
         .
*  THE SRB ROUTINE
         .
         SRBENT   DS   0H
         .
         LR   4,14                     SAVE RETURN ADDRESS
         .
*  ESTABLISH ADDRESSABILITY TO THE DATA SPACE
         .
         USING PARMS,1
         LAM  5,5,DALET                USE REGISTER 5 TO ADDRESS DATA
         SPACE
         .
         SAC   512                     SWITCH INTO AR ADDRESSING MODE
         SYSSTATE ASCENV=AR            SET GLOBAL BIT FOR AR MODE
         .
*  USE DATA SPACE
         .
*  POST THE WAITING TASK
         .
         SAC   0                       SWITCH INTO PRIMARY ADDRESSING
*                                      MODE
SYSSTATE ASCENV=PRIMARY       SET GLOBAL BIT FOR PRIMARY MODE
         .
         POST  ECB1,LINKAGE=SYSTEM
         .
* EXIT
         .
         BR    4
```

Dumping storage in a data space

Use the following macros to dump data space storage.

- Use the DSPSTOR parameter on the **SNAPX macro** to dump storage from any data space that the caller has addressability to, providing the program also has a TCB key (for SCOPE=SINGLE and SCOPE=ALL data spaces) or a PSW key (for a SCOPE=COMMON data space) that matches the storage key of the data space.

- Use the DUMPOPX parameters on the **ABEND macro** and the **SETRP macro** with the list form of the **SNAPX macro** to dump data space storage.
- Use the LISTD and SUMLSTL parameters on the **SDUMPX macro** to dump certain ranges of data space storage:
  - LISTD identifies (by STOKEN) the data space that contains storage to be added to the main part of the dump.
  - SUMLSTL identifies (by ALET) the data space that contains the storage to be added to the summary part of the dump.

For the syntax of SNAPX, see *z/OS MVS Programming: Assembler Services Reference IAR-XCT*. For the syntax of SDUMPX, see *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

Using data spaces efficiently

Although a TCB can own many data spaces, it is important that it reference these data spaces carefully. It is 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. System performance is best if each program completes its work with one data space before it starts work with another data space.

MVS limits the number of access list entries and the number of data spaces available to each TCB. Therefore, given a choice, you must use one large data space rather than a number of small data spaces that add up to the size of the one large data space.

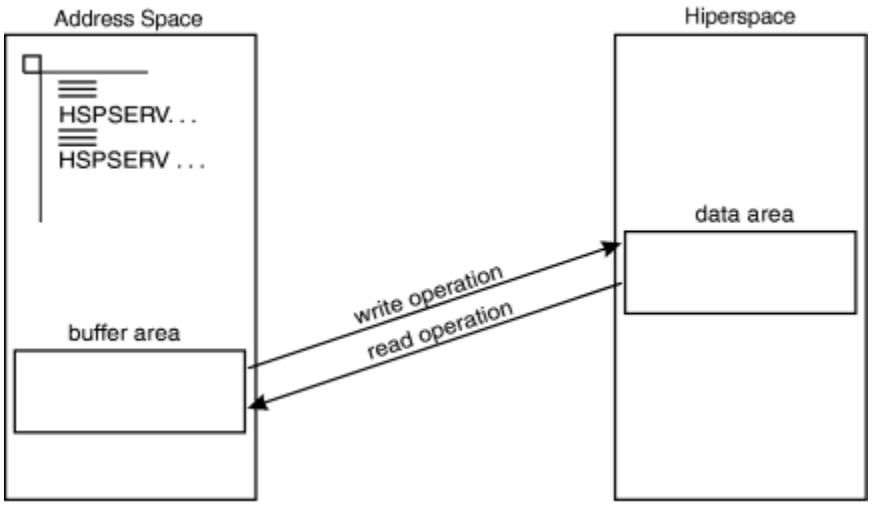
Chapter 7. Creating and using hiperspaces

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 a data space, data 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 its 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 write service performs the transfer of the data to the hiperspace. The HSPSERV read service transfers the hiperspace data back to the address space buffer area.



The data in the hiperspace and the buffer area in the address space must both start on a 4K byte boundary.

A program would use a hiperspace rather than a data space if the program needs an area outside the address space primarily for storage purposes, and not for data manipulation. If you are uncertain whether a hiperspace or a data space is the best choice for your program, see “Basic decision: data space or hiperspace” on page 5.

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, the following books contain the syntax and parameter descriptions for the macros that are mentioned in this section:

- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

Managing hiperspace storage

Managing storage in hiperspaces differs from managing storage in address spaces. Keep the following advisory notes in mind:

- When you create a hiperspace, request a maximum size large enough to handle your application's needs and, optionally, an initial size large enough to meet its immediate needs.
- You are responsible for keeping track of hiperspace storage. You cannot use the system services, such as the STORAGE, GETMAIN, FREEMAIN, or CPOOL macros, or the callable cell pool services to manage this area.
- If you are not going to use an area of a hiperspace again, release that area to free the resources that back the area.
- When you are finished using a hiperspace, delete it.

Limiting hiperspace use

The use of hiperspace consumes system resources such as expanded and auxiliary storage. Programmers responsible for tuning and maintaining MVS can control the use of these resources. Through the system management facility (SMF) installation exit IEFUSI, an installation can set limits on the amount of virtual storage that programs in each address space can use for data space and hiperspace.

See *z/OS MVS Installation Exits* for information on using IEFUSI.

Managing hiperspace storage across a checkpoint/restart operation

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 perform a DSPSERV CREATE TYPE=HIPERSPACE to recreate the hiperspace.
  2. Perform an 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 perform an ALESERV ADD to access the hiperspace.

See *z/OS DFSMSdfp Checkpoint/Restart* for more information about the CHKPT macro.

Relationship between the hiperspace and its owner

Your program creates a hiperspace, but it cannot own the hiperspace. If the unit of work that represents the program is a TCB, that TCB is the owner of the hiperspace unless the program assigns ownership to another TCB. If the unit of work is an SRB, the program must assign ownership to a TCB. Because of this transfer of ownership, the owner of the hiperspace and the creator of the hiperspace are not always the same TCB.

The virtual area of a hiperspace is available to programs that run under the TCB that owns the hiperspace and is available, in some cases, to other programs. When a TCB terminates, the system deletes any hiperspaces the TCB owns. The system swaps a hiperspace in and out as it swaps in and out the address space that dispatched the owning TCB. Thus, hiperspaces that are shared by programs that run in other address spaces must be owned by TCBs in non-swappable address spaces.

A hiperspace can remain active even after the creating TCB terminates. When a program creates a hiperspace, it can assign ownership of the hiperspace to a TCB that will outlive the creating TCB. In this case, the termination of the creating TCB does not affect the hiperspace.

Because hiperspaces belong to TCBs, keep in mind the relationship between the program and the TCB under which the program runs. For simplicity, however, this section describes hiperspaces as if they belong to programs. For example, "a program's hiperspace" means "the hiperspace that belongs to the TCB that represents the program."

Serializing use of hiperspace storage

At many installations, users must share access to data in a hiperspace. Users who are updating data for common use by other programs need exclusive access to that data for the period of time between the transfer of data from the hiperspace to the return of data to the hiperspace. If several users tried to update the same data at the same time, the result would be incorrect or damaged data. To protect the data integrity, you might need to serialize access to the data in the hiperspace.

Serializing the use of the storage in a hiperspace requires similar methods to those you would use to serialize the use of virtual storage in an address space. Use the ENQ and DEQ macros or establish your own protocol for serializing the use of the hiperspace.

Standard and expanded storage only hiperspaces

You have a choice of creating a standard hiperspace or an ESO hiperspace. The **standard hiperspace** 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.

- HSTYPE=SCROLL on DSPSERV creates a standard hiperspace.
- HSPSERV SWRITE and HSPSERV SREAD transfer data to and from a standard hiperspace.

The **ESO hiperspace** is backed with expanded storage only. It is a high-speed buffer area, or "cache" for data that your program needs.

- HSTYPE=CACHE on DSPSERV creates an ESO hiperspace.
- HSPSERV CWRITE and HSPSERV CREAD transfer data to and from an ESO hiperspace.

Standard hiperspace

Standard hiperspace is available to all programs. The data in a standard hiperspace is predictable; that is, your program can write data out 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 50 on page 162 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.

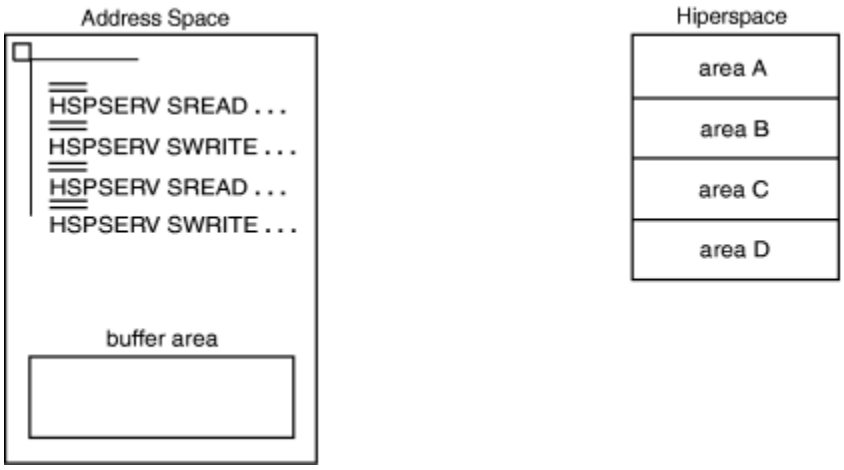


Figure 50. Example of scrolling through a standard hiperspace

A standard hiperspace gives your program an area where it can:

- Store data, either generated by your program or moved (through 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.

Standard hiperspaces can be **non-shared** and **shared**, depending on how you code the SHARE parameter on DSPSERV.

- Generally, a program can access a non-shared standard hiperspace only if it is dispatched in the owner's home address space. However, a program not dispatched in the owner's home address space and using an access list entry token (ALET) can access a non-shared standard hiperspace through the owner's home primary address space access list (PASN-AL).
- A program can share a shared standard hiperspace with programs that are dispatched in any address space.

You can extend the use of hiperspaces by supplying an ALET on the HSPSERV macro. To learn the differences between non-shared and shared standard hiperspaces and how you can extend their use, see “Accessing hiperspaces” on page 169.

If your application wants to save a permanent copy of the data from a standard hiperspace, consider using the services of data-in-virtual. See “Using data-in-virtual with standard hiperspaces” on page 192.

Expanded storage only hiperspaces

An ESO hiperspace is available to supervisor state programs or problem state programs with PSW keys 0 through 7. To use the hiperspace, a program must have the STOKEN for the hiperspace. An ESO hiperspace is backed by expanded storage only. To back this storage, the system does not use auxiliary storage slots; data movement does not include paging I/O operations. However, in a peak-use situation:

- The system might not be able to back the data you are writing to the hiperspace.
- The system might take away the expanded storage that backs the hiperspace.

These actions cause the data in an ESO hiperspace to be volatile. Therefore, use an ESO hiperspace only if you are prepared to handle unsuccessful read operations. You can use this hiperspace to get quick access to the data there. But, in a peak-use condition, when the system takes the expanded storage away from the hiperspace, the program must be prepared to read data from a permanent backup copy on DASD or recreate the data that was in the hiperspace.

When the system swaps the address space out, it discards the data in any hiperspace that is owned by TCBs that are running in the address space. For this reason, you might consider making such an address space non-swappable.

Summary of the differences

Table 16 on page 163 shows some important differences between standard (both non-shared and shared) hiperspaces and ESO hiperspaces:

Table 16. Comparison of standard and ESO hiperspaces

Question	Standard hiperspace	ESO hiperspace
What authorization do you need to create the hiperspace?	Any, for non-shared; supervisor state or PSW key 0-7 for shared.	Supervisor state or PSW key 0-7
What authorization do you need to use the hiperspace?	Any, for non-shared; depends on use of an ALET for shared.	Supervisor state or PSW key 0-7
How do you write data to the hiperspace?	By using HSPSERV SWRITE	By using HSPSERV CWRITE
How do you read data from the hiperspace?	By using HSPSERV SREAD	By using HSPSERV CREAD
Does the system save the data in the address space buffer after a write operation?	No	Yes, unless you use KEEP=NO on HSPSERV
Does the system save the data in the hiperspace after a read operation?	Yes, unless you use RELEASE=YES on HSPSERV	Yes (although hiperspace data is always volatile)
What happens to the data in the hiperspace when the system swaps the owning address space out?	The system preserves the data.	The system discards the data.

Rules for creating, deleting, and using hiperspace

To protect data spaces from unauthorized use, the system uses certain rules to determine whether a program can create, delete, or extend a hiperspace or whether it can access data in a hiperspace. The rules for problem state programs with PSW key 8 through F differ from the rules for programs that are supervisor state or PSW key 0 through 7. Table 17 on page 164 summarizes these rules:

Table 17. Creating, deleting, and using hiperspace			
Function	**Type of hiperspace**	**A problem state, key 8 - F program:**	**A supervisor state or key 0-7 program:**
CREATE	Non-shared standard	Can create a non-shared standard hiperspace.	Can create the hiperspace if its primary or home address space is the same as the intended owner's home address space.
	Shared standard and ESO	Cannot create shared or ESO hiperspaces.	Can create the hiperspace if its primary or home address space is the same as the intended owner's home address space.
DELETE	Non-shared standard	Can delete the non-shared standard hiperspaces it owns if its PSW key matches the storage key of the hiperspace.	Can delete a non-shared standard hiperspace if its primary or home address space is the same as the owner's home address space.
	Shared standard and ESO	Cannot delete a shared standard or ESO hiperspace.	Can delete the hiperspace if its primary or home address space is the same as the owner's home address space.
RELEASE	Non-shared standard	Can release storage in its non-shared standard hiperspaces if its PSW key matches the storage key of the hiperspace.	Can release storage in a non-shared standard hiperspace if its primary or home address space is the same as the owner's home address space and its PSW key matches the storage key of the hiperspace.
	Shared standard and ESO	Cannot release storage in a shared standard or ESO hiperspace.	Can release storage in the hiperspace if its PSW key matches the storage key of the hiperspace.
EXTEND	Non-shared standard shared standard and ESO	Can extend the current size only if it owns the hiperspace.	Can extend the current size.

Creating a hiperspace

To create a hiperspace, issue the DSPSERV CREATE macro with the TYPE=HIPERSPACE parameter. MVS gives you contiguous 31-bit virtual storage of the size you specify and initializes the storage to hexadecimal zeroes. The entire hiperspace has the storage key that you request, or, by default, the key that matches your own PSW key. Use the HSTYPE parameter to specify whether the hiperspace is to be standard or ESO. If standard, you can use the SHARE parameter to request either a non-shared standard (SHARE=NO, the default) or a shared standard (SHARE=YES) hiperspace. If you omit both HSTYPE and SHARE, you create a non-shared standard hiperspace.

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. See “Choosing the name of the hiperspace” on page 165.
- 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. See "Specifying the size of the hiperspace" on page 166.
- A location where DSPSERV can return the address (either 0 or 4096) of the first available block of the hiperspace (ORIGIN parameter). See "Identifying the origin of the hiperspace" on page 168.
- A request that the hiperspace not be fetch-protected (FPROT parameter). See "Protecting hiperspace storage" on page 167.
- A request that the hiperspace be shared standard (SHARE parameter). See "Creating a non-shared or shared standard Hiperspace" on page 168.
- The storage key of the hiperspace (KEY parameter). Use CALLERKEY to specify that the storage key of the hiperspace is to match your PSW key (or take the default for the KEY parameter). See "Protecting hiperspace storage" on page 167.
- The TTOKEN of the TCB to which you assign ownership of the hiperspace (TTOKEN parameter). See "How SRBs use hiperspaces" on page 196.
- A request that the system persist in trying to keep the data in an ESO hiperspace (CASTOUT=NO). See "Creating an expanded storage only Hiperspace" on page 168.

Choosing the name of the hiperspace

The names of hiperspaces and data spaces must be unique within an address space. You can choose the name yourself or you can ask the system to generate a unique name for the hiperspace. 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 *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

Use the GENNAME parameter on DSPSERV to ask the system to generate a unique name for your hiperspace. 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 hiperspace you are creating, use the OUTNAME parameter.

Note that the system has a supply of 99,999 names it can generate for data spaces and hiperspaces for a single address space. If the system tries to generate a name and finds that it has used up the supply of names, it rejects the program with a return code of "08" and a reason code of "0012". The system restores the supply of names whenever the number of such data spaces and hiperspaces owned by the address space goes to zero. Therefore, if your program is a batch job and it is creating a hiperspace, do not:

- Request that the system generate a name (through the GENNAME parameter), ***and***

- Assign ownership to an initiator task or a task higher than the initiator task in the TCB chain

Specifying the size of the hiperspace

When you create a 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.) 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 the control an installation has on the size of 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 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 hiperspace with the default size. (The IBM default size is 239 blocks.)

If you create the hiperspace with a storage key of 0 through 7, the system does not check the size against the total storage already used for data spaces and hiperspaces. If you create the hiperspace with a storage key of 8 through F, the system adds the initial size of the space to the cumulative total of all data spaces and hiperspaces for the address space and checks this total against the installation limit for an address space.

For information on the IBM defaults and how to change them, see “Limiting hiperspace use” on page 160.

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 hiperspace.
- An initial size identifies the amount of the storage you will immediately use.

As you need more space in the hiperspace, you can use the DSPSERV EXTEND macro to increase the size of the available storage, thus increasing the storage in the hiperspace that is available for the program. The amount of available storage is called the **current size**. (At the creation of a 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 of the hiperspace.

If you know the default size and want a hiperspace smaller than or equal to that size, use BLOCKS=maximum size or omit the BLOCKS parameter.

If you know what size 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 hiperspace with the maximum size and the initial size both set to the default size.

If you do not know how large a hiperspace (with storage key 8 - F) 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 hiperspace it creates for you. You would use NUMBLKS, for example, if you did not specify BLOCKS and do not know the default size.

Figure 51 on page 167 shows an example of using the BLOCKS parameter to request a hiperspace with a maximum size of 100,000 bytes of space and a current size of 20,000 bytes.

```
         DSPSERV CREATE,. . .,BLOCKS=(HSMAX,HSINIT)
         .
HSMAX    DC  A((1000000+4095)/4096)         HIPERSPACE MAXIMUM SIZE
HSINIT   DC  A((20000+4095)/4096)           HIPERSPACE INITIAL SIZE
```

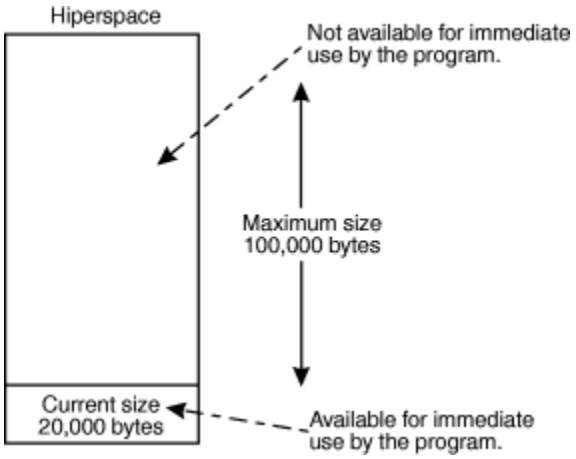


Figure 51. Example of specifying the size of a hiperspace

Figure 61 on page 191 shows how you can extend the available storage of the hiperspace in Figure 51 on page 167.

Protecting hiperspace storage

If a supervisor state or PSW key 0 - 7 program wants the user of the hiperspace to have read-only access, it can use the FPROT and KEY parameters on DSPSERV. KEY assigns the storage key for the hiperspace, and FPROT specifies whether the storage in the hiperspace is to be fetch-protected. Storage protection and fetch protection rules apply for the entire hiperspace. For example, a program cannot reference storage in a fetch-protected hiperspace without holding the PSW key that matches the storage key of the hiperspace or PSW key 0.

Figure 52 on page 168 shows an ESO hiperspace, HSX, with a storage key of 5, owned by a subsystem. PGM1 with PSW key of 8 has access to the hiperspace; however, its PSW key does not match the storage key of the hiperspace. Its ability to access the hiperspace depends on how the creating program coded the FPROT parameter on the DSPSERV macro.

- If the creating program specified no fetch-protection (FPROT=NO), PGM1 can fetch from (using HSPSERV CREAD) but not store into the hiperspace (using HSPSERV CWRITE).
- If the creating program specified fetch-protection (FPROT=YES), PGM1 can neither fetch from nor store into the hiperspace.

In Figure 52 on page 168, PGM1 has fetch and store capability to the hiperspace; the subsystem provides a PC routine with a PSW key 5 in the common area. To access the hiperspace, PGM1 can PC to the PC routine and have access to the hiperspace through the HSPSERV read and write operations. In the same way, other programs can PC to the PC routine and use the data in the hiperspace.

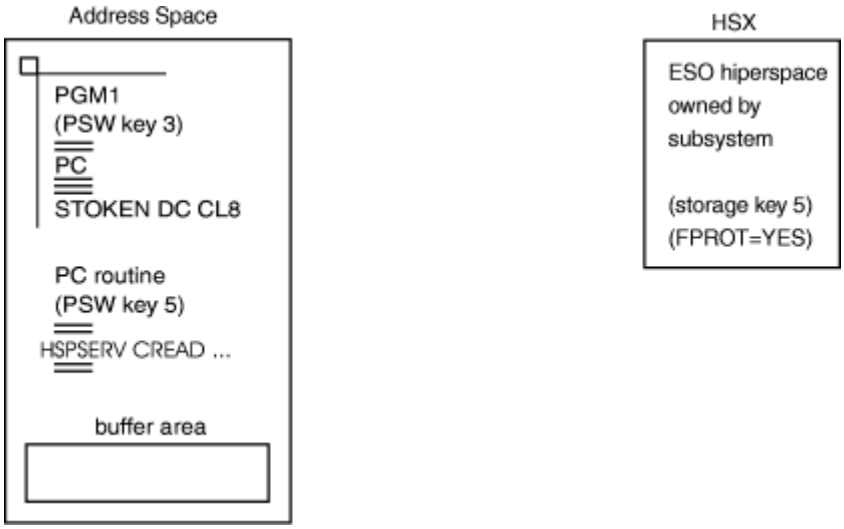


Figure 52. Protecting storage in a hiperspace

Identifying the origin of the hiperspace

Some processors do not allow the hiperspace to start at address zero; these hiperspaces start at the address 4096. When you use DSPSERV CREATE, you can count on the origin of the data space staying the same within the same IPL. To learn the starting address, either (1) create a hiperspace of 1 block of storage more than you need and then assume that the hiperspace starts at 4096 or (2) use the ORIGIN parameter. If you use ORIGIN, the system returns the beginning address of the hiperspace to the location you specify.

An example of the problem you want to avoid in addressing hiperspace storage is described as follows:

> Suppose a program creates a hiperspace of 1 megabyte and assumes the data starts at zero when it really begins at 4096. Then, if the program used the address zero in the hiperspace, the system abends the program.

Creating a non-shared or shared standard Hiperspace

The HSTYPE parameter tells the system which kind of hiperspace you want to create. HSTYPE=SCROLL identifies the standard hiperspace, the kind of hiperspace that your program can **scroll** through. HSTYPE=SCROLL is the default.

SHARE=YES specifies a shared standard hiperspace; SHARE=NO, the default, specifies a non-shared standard hiperspace.

Example of creating a standard Hiperspace

The following example creates a non-shared standard hiperspace:

```
*
         DSPSERV CREATE,NAME=HSNAME,TYPE=HIPERSPACE,HSTYPE=SCROLL,   X
               BLOCKS=20,STOKEN=HSSTOKEN,ORIGIN=HSORG
*
HSNAME   DC   CL8'SCROLLHS'         * NAME FOR THE HIPERSPACE
HSSTOKEN DS   CL8                   * STOKEN OF THE HIPERSPACE
HSORG    DS   F                     * HIPERSPACE ORIGIN
```

Creating an expanded storage only Hiperspace

HSTYPE=CACHE tells the system that your program (in supervisor state or PSW key 0 - 7) wants an ESO hiperspace, the kind of hiperspace that offers your program a high-speed **cache** area.

The CASTOUT parameter on DSPSERV is available only for ESO hiperspaces. This parameter gives the system some indication of the priority of the data in the hiperspace. CASTOUT tells the system how hard it should try to keep the data in the hiperspace. The system looks at this parameter when it makes the decision to take the expanded storage backing from the hiperspace. If you use CASTOUT=NO, the system persists in trying to keep the hiperspace data. It tells the system to give the hiperspace higher priority when it searches for pages to remove from expanded storage when a shortage arises.

Note that specifying CASTOUT=NO can place a heavy demand on expanded storage and does not always protect the data. Use it only when you are willing to sacrifice overall system performance to have better availability of the data. Certain factors might cause the pages to be discarded regardless of CASTOUT=NO. For example, if the system swaps out the address space that owns the hiperspace, it discards pages without regard to CASTOUT. To prevent loss of data due to a swapped-out address space, make the address space that owns the hiperspace non-swappable.

Example of creating an ESO Hiperspace

The following example creates an ESO hiperspace:

```
* CREATE AN ESO HIPERSPACE
         .
DSPSERV1 DSPSERV CREATE,NAME=NAME,STOKEN=STOKEN,ORIGIN=ORIGIN       X
             BLOCKS=BLOCKS,TYPE=HIPERSPACE,HSTYPE=CACHE,CASTOUT=NO
         .
* CONSTANTS AND VARIABLES
         .
NAME       DC  CL8'HSDS01  ' NAME OF THE HIPERSPACE
BLOCKS     DC  F'100'        SIZE OF THE HIPERSPACE IN BLOCKS
ORIGIN     DS  AL4           START ADDRESS OF THE HIPERSPACE
STOKEN     DS  CL8           STOKEN RETURNED FROM DSPSERV CREATE
```

Accessing hiperspaces

The HSPSERV macro service controls the use of a hiperspace. HSPSERV requires that the program that is accessing the hiperspace specify the STOKEN of the hiperspace. The program could have received the STOKEN from DSPSERV or received it from another program. HSPSERV considers the following factors before it allows a program to access a hiperspace:

- The authority of the caller
- The type of hiperspace: non-shared standard, shared standard, or ESO
- Whether the caller is in cross memory mode
- Whether the caller gives an access list entry token (ALET) for the hiperspace to HSPSERV
- On a write operation, whether the PSW key of the accessing program matches the storage key of the hiperspace, or is zero.

When you access hiperspaces, you are not required to use an ALET. However, there are benefits to using ALETs with hiperspaces. By obtaining an ALET, a program builds a connection between the program and a hiperspace. When the program supplies the ALET on HSPSERV, the program can:

- Access some hiperspaces that it could not otherwise access. See "How problem state programs with PSW key 8 through F use a hiperspace" on page 170 and "How supervisor state or PSW key 0 through 7 programs use hiperspaces" on page 172.
- Take advantage of faster or more efficient data transfer. See "Obtaining improved data transfer to and from a hiperspace" on page 180.

A program has two ways to obtain an ALET:

- From another program, as a passed parameter
- From the ALESERV ADD macro, if the program has the STOKEN for the hiperspace.

The decisions of whether to create a non-shared standard or shared standard hiperspace and whether to obtain an ALET depend on how you plan to share the data in the space. The sections that follow help you

understand which hiperspaces HSPSERV allows problem state and supervisor state programs to access, and also the benefits of having an ALET. “Obtaining an ALET for a hiperspace” on page 174 describes how to obtain an ALET.

How an ALET connects a program to a hiperspace

An ALET is an index to an access list. An access list is a table where each entry represents an address space, data space, or hiperspace that programs can access. Each program has two access lists: a primary address space access list (PASN-AL) and a dispatchable unit access list (DU-AL).

Each address space has one PASN-AL. It is available to any program that has that address space as its primary address space.

Each TCB and SRB has one DU-AL. It is available to any program that the TCB or SRB represents.

To use one of these access lists, the program needs the ALET that indexes the access list. It uses the ALET as input on the HSPALET parameter on HSPSERV.

Chapter 5, “Using access registers,” on page 93 describes ALETs for data spaces and address spaces; for an illustration of a DU-AL and PASN-AL, see Figure 27 on page 101. Chapter 8, “Creating address spaces,” on page 197 6 limits its discussion to ALETs for hiperspaces.

How problem state programs with PSW key 8 through F use a hiperspace

A problem state program with PSW key 8 - F can use the non-shared hiperspace it created. Figure 53 on page 170 illustrates this use.

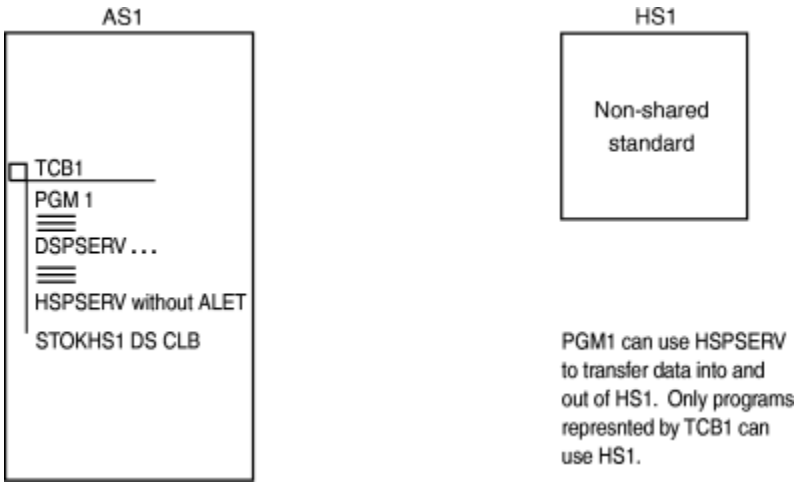


Figure 53. A problem state program using a non-shared standard hiperspace

By obtaining an ALET, a problem state program with PSW key 8 - F obtains the benefit of the move-page facility and also shares a hiperspace with a subtask. For example, suppose a problem state program obtains an ALET, attaches a subtask using the ALCOPY parameter on the ATTACH macro, and passes the ALET and STOKEN to the subtask. These actions allow the task and its subtask to share the same non-shared hiperspace. Two problem state programs can share a SCOPE=SINGLE data space in the same way. Turn to “How problem state programs with PSW key 8 through F use a hiperspace” on page 170 for an such an example.

Can an authorized program set up an environment in which an unauthorized program can share hiperspaces? An authorized program (supervisor state with PSW key 0 - 7) can set up addressability for an unauthorized program (problem state with PSW key 8 - F) and increase the use of hiperspaces by those unauthorized programs. This section contains two examples of this increased capability.

- Example 1 shows an unauthorized program using a shared or non-shared standard hiperspace through an entry on the PASN-AL. Figure 54 on page 171 illustrates the first example.
- Example 2 shows an unauthorized program using a hiperspace while the program is in cross memory mode. Figure 55 on page 172 illustrates the second example.

Example 1 shows how an entry on the PASN-AL allows all programs in the address space, including unauthorized programs, to use either non-shared or shared standard hiperspaces. An authorized program obtains the hiperspace, places an entry on the PASN-AL, and obtains the ALET. The program then passes the ALET and STOKEN to other programs in the address space. Even programs that space-switch into the address space can use the hiperspace, providing they receive the ALET and STOKEN. Figure 54 on page 171 illustrates this case.

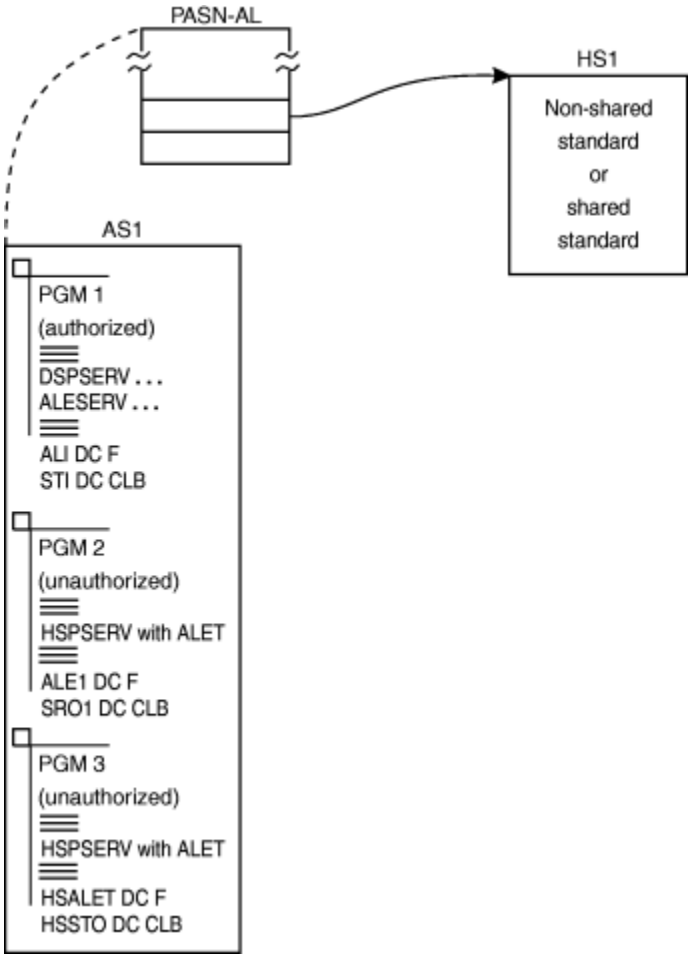


Figure 54. Example 1: An unauthorized program using a standard hiperspace

Example 2 shows how an authorized program can set up a cross memory environment that allows an unauthorized program to space-switch and still have access to a non-shared hiperspace. Having created the non-shared standard hiperspace HS1, PGM1 can obtain an ALET on the DU-AL, space-switch, and use the ALET and STOKEN to access HS1. Figure 55 on page 172 illustrates this use of a non-shared hiperspace.

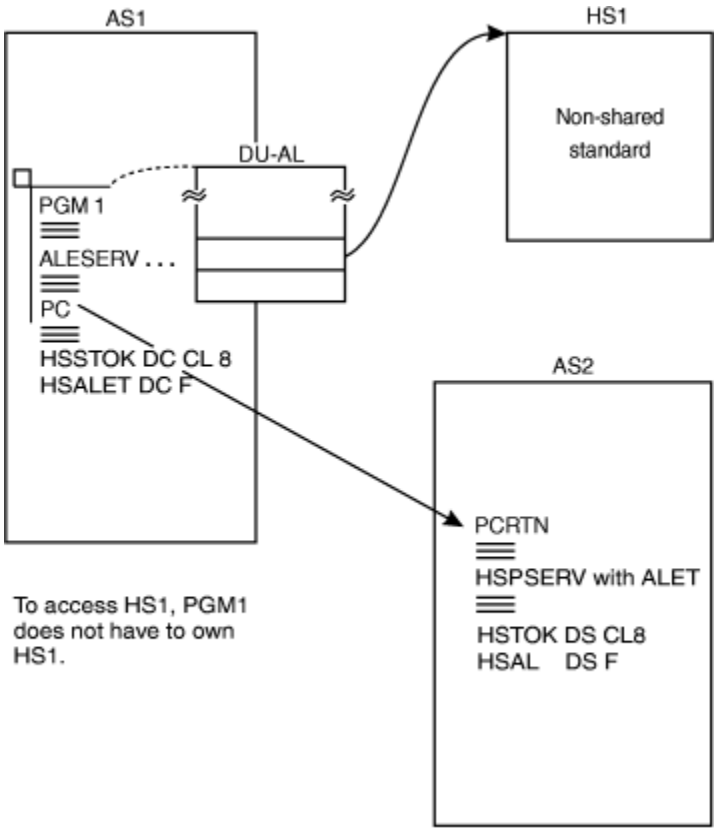


Figure 55. Example 2: An unauthorized program using a standard hiperspace

Summary of unauthorized programs' use of hiperspaces

Table 18 on page 172 describes the rules for accessing hiperspaces for problem state programs with PSW key 8 - F.

Table 18. Hiperspaces that problem state programs with PSW 8 - F can access

If the program does not have an ALET:	**If the program has an ALET:**
It can access a **non-shared standard** hiperspace that it owns. It cannot be in cross memory mode.	It can access a **non-shared standard** hiperspace. It can be in cross memory mode.
It cannot access a **shared standard** hiperspace.	It can access a **shared standard** hiperspace. It can be in cross memory mode.
It cannot access an **ESO** hiperspace.	

How supervisor state or PSW key 0 through 7 programs use hiperspaces

Supervisor state or programs with PSW key 0 - 7 can create and control all three types of hiperspaces. Table 19 on page 173 identifies the three types of hiperspaces that these programs can use and some restrictions on this use.

Table 19. Hiperspaces that supervisor state or PSW key 0 - 7 programs can use	
If the program does not have an ALET:	**If the program has an ALET:**
• It can access a **non-shared standard** hiperspace if the owner's home address space is the same as the program's home address space. The program must not be in cross memory mode. • It can use a **shared standard** or **ESO** hiperspace. It can be in cross memory mode.	• Access to a **non-shared standard** is the same as if the program did not have an ALET, except it can be in cross memory mode. • Access to a **shared standard** or **ESO** hiperspace is the same as if the program did not have an ALET.

The use of an ALET allows supervisor state or PSW key 0 - 7 programs to use non-shared standard hiperspaces. The following section describes how this program can:

- Use a non-shared standard hiperspace.
- Use a shared standard and ESO hiperspace.
-

The supervisor state program using a non-shared hiperspace

Like the problem state program with PSW key 8 - F, the supervisor state program in cross memory mode can use HSPSERV with an ALET to access a non-shared standard hiperspace. For example, in Figure 56 on page 173, PGM1 in AS1 can place an entry for HS1 on the DU-AL and receive the ALET. PGM1 can then PC to AS2, passing the STOKEN and ALET to PCRTN. PCRTN can access HS1.

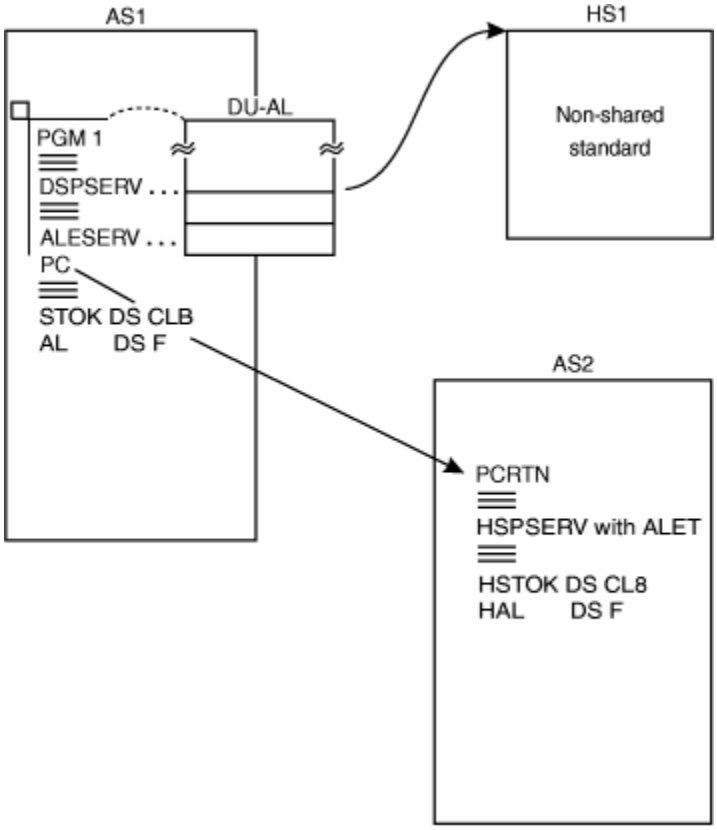


Figure 56. A supervisor state program using a non-shared standard Hiperspace

Note that PCRTN could not access HS1 unless it used HSPALET on HSPSERV.

The supervisor state program using shared standard and ESO hiperspaces

Supervisor state or PSW 0 - 7 programs can access any shared standard or ESO hiperspace, providing they have the STOKEN of the hiperspace. They are not required to have an ALET.

Figure 57 on page 174 illustrates two programs in two address spaces. Both of these programs can access data in HS1 without using the HSPALET parameter. PGM1, the creator of HS1, passes the STOKEN of HS1 to PGM2.

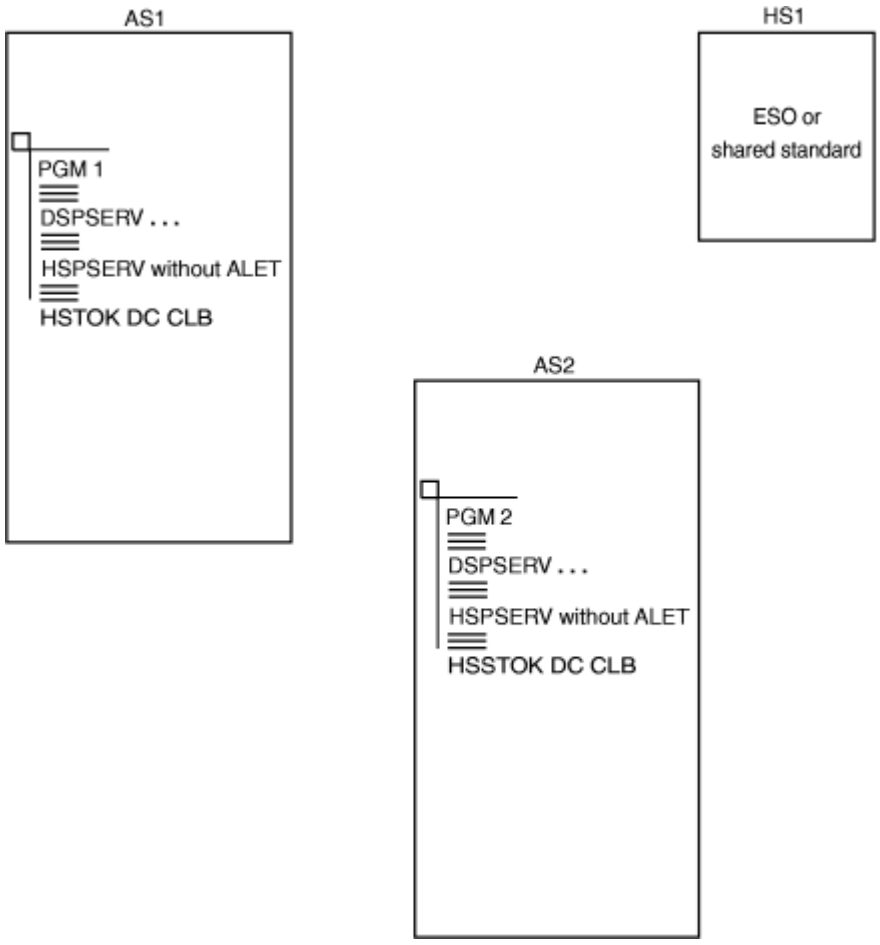


Figure 57. A supervisor state program using a shared standard hiperspaces

With a little more effort, PGM1 and PGM2 in Figure 60 on page 181 could get faster transfer of data to and from expanded storage; "Obtaining improved data transfer to and from a hiperspace" on page 180 describes how to gain the added performance. An example of PGM1 and PGM2 using ALETs on HSPSERV is in that section.

Obtaining an ALET for a hiperspace

Use ALESERV ADD to obtain an ALET and place an entry on a DU-AL or PASN-AL. Whether a program can use ALESERV ADD depends on the authority of the program, the type of hiperspace, and whether the program is in cross memory mode. The rules that apply to the programs that use ALESERV ADD are described in Table 20 on page 175.

Table 20. Rules for adding access list entries for hiperspaces			
Function	**Type of hiperspace**	**A problem state, key 8 - F program:**	**A supervisor state or key 0-7 program:**
Add entries to the DU-AL	Non-shared standard	Can add entries for a hiperspace it owns.	Can add entries if the caller's and owner's home address space is the same.
	Shared standard and ESO	Cannot add entries for the hiperspace to its DU-AL.	Can add entries. For ESO hiperspace, see Programming note 1.
Add entries to the PASN-AL	Non-shared standard	Cannot add entries.	Can add entries if caller's primary address space is the same as the owner's home address space.
	Shared standard and ESO	Cannot add entries.	Can add entries. For ESO hiperspace, see Programming note 1.

Programming notes:

1. Do not add an entry for an ESO hiperspace to any access list that is available to a problem state program with PSW 8 - F. In other words, do not add an entry to a PASN-AL if unauthorized programs will be executing in the address space. Do not add an entry to a DU-AL if its TCB will be representing an unauthorized program.
2. Use ALESERV ADD with hiperspace.
3. The system rejects an ALESERV ADD request if the hiperspace is currently defined as a data-in-virtual object.

Example of adding an access list entry for a hiperspaces

The following code uses DSPSERV to create a non-shared standard hiperspace named TEMP. The system returns the STOKEN of the hiperspace in HSPCSTKN and the origin of the hiperspace in HSPCORG. The ALESERV ADD macro returns the ALET in HSPCALET. The program uses the ALET on the HSPALET parameter on HSPSERV to access the hiperspace.

```
         DSPSERV CREATE,TYPE=HIPERSPACE,NAME=HSPCNAME,
               STOKEN=HSPCSTKN,BLOCKS=HSPBLCKS,ORIGIN=HSPCORG
         ALESERV ADD,STOKEN=HSPCSTKN,ALET=HSPCALET,AL=PASN
         .
HSPCSTKN DS   CL8                     HIPERSPACE STOKEN
HSPCALET DS   F                       HIPERSPACE ALET
HSPCORG  DS   F                       HIPERSPACE ORIGIN RETURNED
HSPCNAME DC   CL8'TEMP    '           HIPERSPACE NAME
HSPBLCKS DC   F'1000'                 HIPERSPACE SIZE (IN 4K BLOCKS)
```

Obtaining and passing ALETs for hiperspaces

To allow other programs to share hiperspaces, a program passes the ALET of the hiperspace to other programs. Because ALETs index into specific access lists, a program can pass:

- An ALET that indexes into an entry on a DU-AL if the passing program and the receiving code are represented by the same TCB
- An ALET that indexes into an entry on a DU-AL if the passing program attached the receiving program (using the ALCOPY parameter on ATTACH or ATTACHX) and passed the entry for the hiperspace on the DU-AL
- An ALET that indexes into the PASN-AL if the ALET indexes into the PASN-AL of the receiving program.

Generally, when two programs in two address spaces share the data in the same hiperspace, the programs must both use ALESERV to add entries to their access lists.

Deleting an access list entry for a hiperspace

Access lists have a limited size; the DU-AL has up to 509 entries and the PASN-AL has up to 510 entries. Therefore, it is a good programming practice to delete entries from an access list when the entries are no longer needed. The specific rules are:

- If a program needs an entry for a short period of time, it should delete the entry when it no longer needs the entry.
- If a program adds an entry and uses that entry during execution, the program does not need to delete the entry; the system deletes the entry when the task terminates.
- Once the entry is deleted, the system can immediately reuse the ALET.

Use ALESERV DELETE to delete an entry on an access list. The ALET parameter identifies the specific entry.

Programs that share hiperspaces with other programs have another action to take when they delete an entry from an access list. They should notify the other programs that the entry is no longer connecting the ALET to the hiperspace. Otherwise, those programs might continue to use an ALET for the deleted entry. See "ALET reuse by the system" on page 114 for more information.

Example of deleting a hiperspace entry from an access list

The following example deletes the entry for the ALET at location HSPCALET. The example also includes the deletion of the hiperspace with a STOKEN at location HSPCSTKN.

```
         ALESERV DELETE,ALET=HSPCALET    REMOVE HS FROM AL
         DSPSERV DELETE,STOKEN=HSPCSTKN  DELETE THE HS
         .
HSPCSTKN DS   CL8                     HIPERSPACE STOKEN
HSPCALET DS   F                       HIPERSPACE 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.

Transferring data to and from a hiperspace

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 transfer the data in 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 58 on page 177 shows three virtual storage areas that you need to identify 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.

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.

- 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.
- NUMRANGE optionally specifies the number of data areas the system is to read or write. The default is one data area.

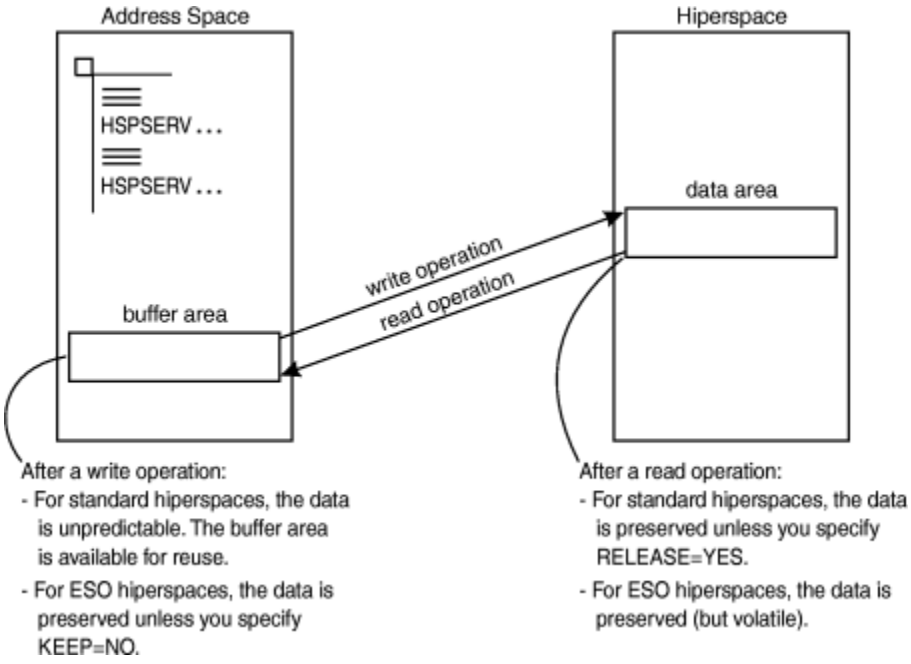


Figure 58. Illustration of the HSPSERV write and read operations

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 *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*. 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 58 on page 177 tells you what the system does with the areas after it completes the transfer. The sections "Read and write operations for standard hiperspaces" on page 177 and "Read and write operations for expanded storage only hiperspaces" on page 179 describe how you use the HSPSERV macro.

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. Note that when a hiperspace is not fetch-protected, HSPSERV SREAD,RELEASE=NO works even when the program's PSW key does not match the storage key of the hiperspace.

To use the HSPSERV macro without an ALET for a non-shared standard hiperspace, the buffer area in the address space must be in the program's home address space. That is, the program cannot be in cross memory mode (where PASN is not equal to HASN).

A program cannot issue a HSPSERV SWRITE to an area of a hiperspace that has a DIV SAVE in progress.

See "Example of creating a standard hiperspace and using It" on page 178 for an example of the HSPSERV SREAD and HSPSERV SWRITE macros.

Example of creating a standard hiperspace and using It

The following example creates a non-shared standard hiperspace named SCROLLHS. The size of the hiperspace is 20 blocks. The program:

- Creates a non-shared 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 59 on page 178 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.

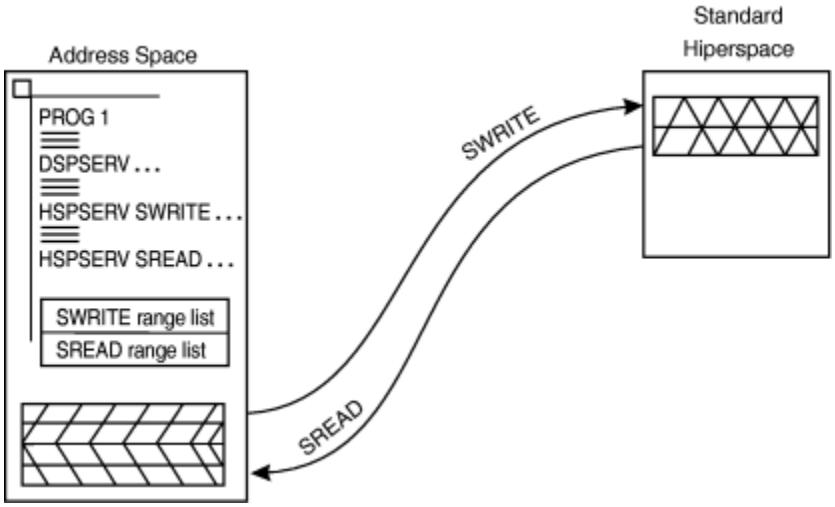


Figure 59. Example of creating a standard hiperspace and transferring data

```
* 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,NUMRANGE=ONE
*
* THE SYSTEM REUSES THE FIRST TWO ADDRESS SPACE PAGES AFTER THE SWRITE
         .
```

```
* 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,NUMRANGE=ONE
*
* 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'
ONE      DC   F'1'                  * NUMBER OF RANGES
         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
```

Read and write operations for expanded storage only hiperspaces

The system backs ESO hiperspaces with expanded storage, a finite resource that many programs compete for. Because an ESO hiperspace is backed with expanded storage, it can be accessed very quickly. However, because of the contention for expanded storage, the data in the hiperspace might not be there when you need it. Because of this uncertainty, your program must have an alternate way to retrieve or recreate the data.

HSPSERV CREAD transfers data from a source location in an ESO hiperspace to an address space. If all blocks requested are available in the hiperspace (that is, are backed by expanded storage) then the system performs the read operation. However, if one or more blocks to be read are no longer available in the hiperspace, then the system rejects the request and returns a failing return code. If the HSPSERV CREAD is successful, the system moves the data to the buffer area in your address space and preserves the data in the source area of the hiperspace, when possible.

HSPSERV CWRITE transfers data from a source location in an address space to a hiperspace. If the system is unable to write all the requested blocks to the hiperspace (because of a shortage of expanded storage), then it rejects the request. In this case, the data in the target area of the hiperspace is volatile. **After the system rejects a HSPSERV CWRITE request, do not issue HSPSERV CREAD using that target area as the source for the CREAD until you have successfully completed a HSPSERV CWRITE to the same area.**

You can request that the system preserve the source data in the address space after it successfully completes the HSPSERV CWRITE operation. If your program will use this same source data again, specify KEEP=YES on HSPSERV (or use the default). KEEP=NO tells the system that you will not be using the source data again. In this case, the system can reuse the pages that back the address space buffer area. In most cases, KEEP=NO gives your program better performance than KEEP=YES.

To use the HSPSERV macro for an ESO hiperspace, the buffer area that is the source of the CWRITE and the target of the CREAD can be in the caller's home address space as well as the caller's primary address space or the common storage area (CSA). This flexibility means that the caller can use the HSPSERV macro while in cross memory mode (that is, where PASN is not equal to HASN).

The following example shows a program transferring data to and from an ESO hiperspace. The address space has one buffer area to receive the hiperspace data. For an example of storing information into the range list, see "Example of creating a standard hiperspace and using It" on page 178.

```
* GENERATE DATA AND WRITE IT TO THE HIPERSPACE
          .
* BUILD RANGE LIST AND PLACE POINTER TO ADDRESS OF LIST IN RANGPTR
          .
          HSPSERV CWRITE,ADDRSP=HOME,STOKEN=HSSTOK,RANGLIST=RANGPTR,    X
              RETCODE=SRVRCODE,RSNCODE=SRSNCODE
          .
```

```
* READ FROM THE HIPERSPACE, IF EVERYTHING HAS BEEN SUCCESSFUL
          .
          HSPSERV CREAD,STOKEN=HSSTOK,RANGLIST=RANGPTR,                 X
              RETCODE=SRVRCODE,RSNCODE=SRSNCODE
          .
```

```
HSSTOK     DS  CL8           STOKEN RETURNED FROM DSPSERV CREATE
RANGPTR    DC  A(RANGLIST)   ADDRESS OF RANGLIST PARM AREA
RANGLIST   DS  0CL12
ASLOC      DS  AL4           ADDRESS OF START OF ADDRESS SPACE AREA
HSLOC      DS  AL4           ADDRESS OF HIPERSPACE AREA TO WRITE TO/FROM
NUMBLKS    DS  F             NUMBER OF BLOCKS TO READ/WRITE
SRVRCODE   DS  F             RETURN CODE
SRSNCODE   DS  F             REASON CODE
```

Obtaining improved data transfer to and from a hiperspace

By specifying the HSPALET parameter on the HSPSERV macro, a program can get faster data movement between central storage and expanded storage. If the data identified on HSPSERV is in expanded storage, HSPSERV uses the move-page facility. If the data is in auxiliary storage, the data transfer still occurs, but without using the move-page facility.

Through the IOSADMF macro, a program can use ADMF to get more efficient data movement between central and expanded storage. Data transfer with the ADMF might be more efficient depending on the number of pages of data you want to transfer.

The IOSADMF macro provides a programming interface to the ADMF. IOSADMF can be used with standard and ESO hiperspaces. With IOSADMF, programs that buffer large amounts of storage in hiperspaces become more efficient because of reduced overall processor use. Processor cycles previously used to move data now become available for the system or other programs to use.

Programs that want to reduce processor time for buffer management, but that find the response time associated with I/O buffering unacceptable, will find the IOSADMF service particularly useful. However, IBM recommends that you design programs that move data to use either the ADMF (IOSADMF macro) or the move-page facility (HSPSERV macro) for the following reasons:

- You cannot use IOSADMF to transfer data unless data already is stored in the hiperspace. Therefore, under certain circumstances, you must use HSPSERV before using IOSADMF to transfer data.
- If the ADMF is not available, your program can attempt the data transfer again by issuing HSPSERV.
- If your program moves variable amounts of data, you might want to design your program to determine which facility best matches each data transfer request.

Which facility best matches your request depends on the number of pages you want to transfer. When the ADMF is available and the program issues IOSADMF to move data, the system determines which facility is appropriate by comparing the number of pages with a system-specific value. If you want to know what that value is so your program can determine which facility to use, issue IOSADMF with the CROSSOVER parameter, and the system returns the value to you.

z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG describes the HSPSERV macro and the IOSADMF macro.

The move-page facility

Note: The move-page facility is standard in all releases of z/OS.

As an example of using the HSPSERV macro with the HSPALET parameter, suppose PGM1 and PGM2 that are illustrated in Figure 57 on page 174 want to gain fast transfer of data to and from expanded storage. Both programs use ALESERV ADD to add entries to access lists; PGM1 adds to its PASN-AL and PGM2 adds to its DU-AL. They use HSPALET on HSPSERV. The HSPALET parameter is available to both problem state programs and supervisor state programs. Figure 60 on page 181 describes this scenario.

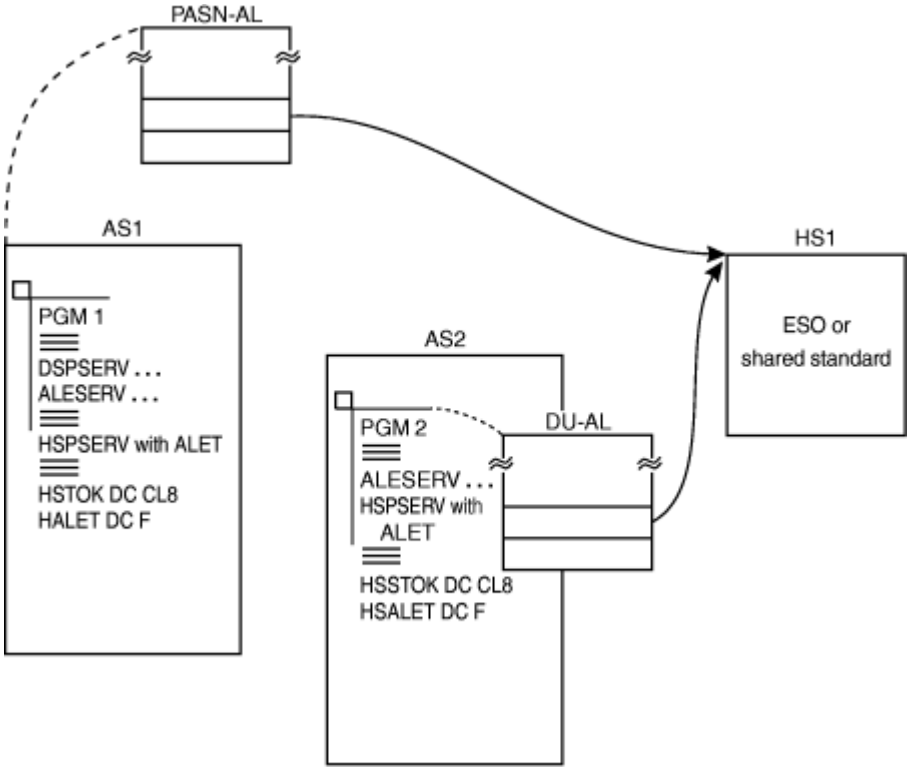


Figure 60. Gaining fast data transfer to and from expanded storage

Before you issue the HSPSERV macro with the HSPALET parameter, place the address of a 144-byte work area in GPR 13 and zero in AR 13.

Restrictions on the combined use of hiperspaces and data-in-virtual are listed in "Using data-in-virtual with standard hiperspaces" on page 192.

Example of an HSPSERV with additional performance: The following example shows a program creating a non-shared standard hiperspace. To get additional performance from HSPSERV, the program obtains the ALET from the ALESERV macro and uses the 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,HSTYPE=SCROLL,     X
               SHARE=NO,BLOCKS=20,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 THE ADDRESS SPACE STORAGE, AND DELETE THE HIPERSPACE
   .
```

```
* DATA AREAS AND CONSTANTS
   .
HSNAME   DC    CL8'SCROLLHS'          * NAME FOR THE HIPERSPACE
HSSTOKEN DS    CL8                    * STOKEN FOR THE HIPERSPACE
HSALET   DS    CL4                    * ALET FOR THE HIPERSPACE
ASPTR    DS    1F                     * LOCATION OF ADDR SPACE STORAGE
SAVER13  DS    1F                     * LOCATION TO SAVE GPR 13
SAVEAR13 DS    1F                     * LOCATION TO SAVE AR 13
WORKAREA DS    CL144                  * WORK AREA FOR HSPSERV
ONEBLK   DC    F'4096'                * LENGTH OF ONE BLOCK OF STORAGE
TWOBLKS  DC    F'8092'                * LENGTH OF TWO BLOCKS OF STORAGE
SRCTEXT1 DC    CL20' INVENTORY ITEMS    '
SRCTEXT2 DC    CL20' INVENTORY SURPLUSES'
         DS    0F
RANGPTR1 DC    A(SWRITLST)            * ADDRESS OF SWRITE RANGE LIST
RANGPTR2 DC    A(SREADLST)            * ADDRESS OF SREAD RANGE LIST
         DS    0F
SWRITLST DS    0CL12                  * SWRITE RANGE LIST
ASPTR1   DS    F                      * START OF ADDRESS SPACE SOURCE
HSORIG1  DS    F                      * TARGET LOCATION IN HIPERSPACE
NUMBLKS1 DC    F'2'                   * NUMBER OF 4K BLOCKS IN SWRITE
         DS    0F
```

```
SREADLST DS    0CL12                  * SREAD RANGE LIST
ASPTR2   DS    F                      * TARGET LOCATION IN ADDR SPACE
HSORIG2  DS    F                      * START OF HIPERSPACE SOURCE
NUMBLKS2 DC    F'2'                   * NUMBER OF 4K BLOCKS IN SREAD
         DS    0F
```

Using the ADMF

To use ADMF support for hiperspaces, design your program to do the following:

1. Determine if the ADMF is available by issuing the IOSADMF macro with the QUERY parameter.
2. Create a hiperspace by issuing the DSPSERV macro with the CREATE parameter.

   Use caution if you specify CASTOUT=NO when creating a hiperspace. Because CASTOUT=NO discourages the system from reclaiming expanded storage areas, the amount of expanded storage available for system use is decreased.
3. Obtain an ALET associated with the hiperspace by issuing the ALESERV macro with the STOKEN received from issuing the DSPSERV CREATE.
4. Put data in the hiperspace by issuing the HSPSERV macro.

   You cannot use IOSADMF to transfer data until the hiperspace already contains data. If the hiperspace storage gets reclaimed, you must add data to the hiperspace again with HSPSERV. (If the hiperspace is a standard hiperspace, HSPSERV will retrieve data from auxiliary storage to refresh the hiperspace data.)
5. Use the IOSADMF macro with the hiperspace's ALET to transfer data to and from the hiperspace.

   The IOSADMF macro AREAD request transfers data from the hiperspace to the program's primary address space. The IOSADMF macro AWRITE request transfers data from the user's primary address space to the hiperspace. If the IOSADMF macro is issued before data is added to the hiperspace, the IOSADMF request will fail. A program cannot issue an IOSADMF macro AWRITE request to an area of a hiperspace that has a DIV SAVE in progress.
6. If your program receives a return code 4, you can try the same operation using the HSPSERV macro. When designing your program to use IOSADMF, check the specific actions for the IOSADMF return and reason code in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* to determine when you can attempt the same operation using HSPSERV instead of IOSADMF.

More than one IOSADMF request can be active for a hiperspace. When you have more than one active IOSADMF request, keep track of the requests and ensure that all data transfer is complete before deleting the hiperspace. If there are outstanding active requests and you issue DSPSERV DELETE for a hiperspace, your program will abnormally end and the hiperspace will not be deleted. If you cannot determine whether all outstanding IOSADMF requests have completed, you can issue IOSADMF APURGE to stop any outstanding requests.

IOSADMF APURGE should be used **only** as a last resort because of data integrity concerns. IOSADMF APURGE immediately stops data transfer for every outstanding IOSADMF request for the specified hiperspace, regardless of the state of those data transfers, and abnormally ends any active operation to the hiperspace. If a new request is subsequently started for the specified hiperspace, the request will process.

If you have a single unit of work that creates the hiperspace, issues the IOSADMF requests, and deletes the hiperspace, you do not have to be concerned about outstanding requests; the system completes the data transfer before processing the delete request. The following is an example of data transfer using the ADMF.

```
*********************************************************************
*02* Description = Sample program to illustrate how to use the       *
*                  IOSADMF services.                                  *
*                                                                     *
*                                                                     *
*02* Function =                                                      *
*                                                                     *
*              ADMFSAMP does not perform any useful work;             *
*              however it does illustrate how the IOSADMF             *
*              services are used.  ADMFEXMP will create a             *
*              hiperspace, create an address space buffer,            *
*              initialize the address space buffer, use               *
*              HSPSERV to write data to the hiperspace                *
*              from the address space buffer, and then                *
*              use IOSADMF to read data from the hiperspace           *
*              back into the address space buffer.                    *
*              In more detail, here is what ADMFEXMP does:            *
*               - Changes mode to supervisor state key 0.             *
*                 IOSADMF requires the caller to be authorized.       *
*               - Obtains a dynamic area.  ADMFSAMP is                *
*                 reentrant and therefore requires a                  *
*                 dynamic area.  ADMFSAMP was written                 *
*                 as a reentrant routine for illustration             *
*                 and independence of caller's mode or                *
*                 key.                                                *
*               - Determines if ADMF is available on the              *
*                 current machines by issuing IOSADMF with            *
*                 the AQUERY service.                                 *
*               - Creates a hiperspace                                *
*               - Obtains an address space buffer area and            *
*                 clears it.                                          *
*               - Initializes the address space buffer area           *
*               - Writes the address space buffer pages to            *
*                 the hiperspace                                      *
*               - Clears the address space buffer area                *
*               - Fixes the page address space storage                *
*               - Uses IOSADMF to read the hiperspace pages           *
*                 back into the address space storage areas           *
*               - Cleans up resources and returns to caller           *
*                                                                     *
*              Again, this routine is only for illustration           *
*              purposes.                                              *
*                                                                     *
*                                                                     *
```

```
*      SECURITY NOTICE = This sample should be used ONLY on a test    *
*                        system.  It does not contain authorization   *
*                        checking required for running on a           *
*                        production system.                           *
*                                                                     *
*      ENVIRONMENT:  AMODE = 31                                       *
*                    RMODE = 31                                       *
*                    STATE = SUPERVISOR                               *
*                    KEY   = 0                                        *
*                    RENT  = YES                                      *
*                                                                     *
*                                                                     *
```

```
*      INPUT:  NONE                                                 *
*                                                                   *
*      REGISTER USAGE:                                              *
*               R9  BASE REGISTER FOR LOAD MODULE                   *
*               R6  POINTS TO DYNAMIC AREA                          *
*           ALL OTHERS STANDARD USAGE                               *
*.................................................................*
         EJECT
ADMFEXMP CSECT
ADMFEXMP AMODE 31                    31-BIT ADDRESSING MODE
ADMFEXMP RMODE ANY                   Rmode any
         SPACE 1
*.................................................................*
*         REGISTER ASSIGNMENTS                                    *
*.................................................................*
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6                     Dynamic area register
R7       EQU   7
R8       EQU   8
R9       EQU   9                     Module base register
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         SPACE 3
         TITLE 'ADMFEXMP -  ADM Sample for AQUERY'
*.................................................................*
*                                                                 *
*        Standard Entry Linkage                                   *
*                                                                 *
*.................................................................*
         PRINT GEN
         USING *,R9              Sets up base reg
         LR    R9,R15            Establish module base
ENTRY    STM   R14,R12,12(R13)   Save caller's regs
         MODESET KEY=ZERO,MODE=SUP
         LA    R0,DYNSIZE        Load length of dynamic area
         STORAGE OBTAIN,LENGTH=((R0)),SP=233 Gets dynamic area
         LR    R6,R1             Gets dynamic area address
```

```
         USING DYNAREA,R6        Sets up dynamic area
         ST    R13,SAVEBK        Save caller's save area addr
         ST    R6,SAVEFW         Save ADMFEXMP save area address
         B     MAINLINE
         DC    CL8'ADMFEXMP'
         DC    CL8'&SYSDATE'
         DC    CL8'&SYSTIME'
         TITLE 'ADMFEXMP - ADMF mainline '
*.................................................................*
*                                                                 *
*        MAINLINE                                                 *
*                                                                 *
*.................................................................*
MAINLINE DS   0H
              L     10,X'10'            Load CVT pointer
              USING CVT,10
              TM   CVTDCB,CVTOSEXT     Is the OSLEVEL extension present
              BNO  NO_ADMF              No, pre-MVS/SP Version 3 system
*
              TM   CVTOSLV1,CVTH4430    Running on version HBB4430?
              BNO  NO_ADMF              No, pre-HBB4430 system.  ADMF
*                                        supported on HBB4430 and above
*
*             Running on HBB4430 system.  Must determine if ADMF
*             software and hardware have been installed on this
*             processor
*
*.................................................................*
*                                                                 *
*        Issues the IOSADMF macro with the AQUERY parameter       *
*        to determine if the ADMF hardware and software are       *
*        available.                                               *
*                                                                 *
*        Note - the IOSADMF macro may be issued on an HBB4430     *
```

```
*        system, however, the full support for ADMF requires          *
*        the ADMF PTF as well as hardware support.                    *
*                                                                     *
*.....................................................................*
              IOSADMF AQUERY,                                         X
               CROSSOVER=CROSSOVER_#,                                 X
               RETCODE=ADMF_INSTALLED_RC,                             X
               RSNCODE=REASONCODE,                                    X
               MF=(E,ADMFLIST)
              L R15,ADMF_INSTALLED_RC    Obtains return code from
*                                        parameter list
              LTR R15,R15                Test for 0 return code
              BNZ  NO_ADMF
ADMF_INSTALLED DS   0H                       ADM support is available
*.....................................................................
*
*  ADMF is installed and available.  Begin function processing to
*   illustrate how ADMF is used
*
*.....................................................................
*
              BAL      R14,CREATE_HS     Build Hiperspace
              LTR      R15,R15           Test for 0 return code
              BNZ      EXIT              Exit if bad RC
```

```
              BAL      R14,AS_STORAGE    Get address space storage
              BAL      R14,INIT_AS       Initialize addr space storage
              BAL      R14,ISSUE_HSPSERV Initialize HS space storage
              LTR      R15,R15           Test for 0 return code
              BNZ      EXIT              Exit if bad RC
              BAL      R14,ISSUE_IOSADMF Use ADMF to read data from HS
*.....................................................................
* Begins clean up operations
*.....................................................................
              DSPSERV  DELETE,STOKEN=HSSTOKEN
              LR      R3,R1              Loads area addr in R3
              A       R3,LENGTH_AS_AREA  Adds length of area to addr
              BCTR    R3,0               Subtracts 1 to get end addr
              PGSER R,FREE,A=(R1),EA=(R3),ECB=0
              L        R3,LENGTH_AS_AREA
              STORAGE RELEASE,SP=229,ADDR=ASPTR,LENGTH=(R3)
              B        EXIT
NO_ADMF       DS       0H
*.....................................................................
*
*    ADM support is either not installed or not
*    available on this release
*
*.....................................................................
         WTO   'ADMFEXMP - ADMF not installed. Sample ends',          X
               ROUTCDE=(11),DESC=(2)
                LA    R3,12                  Loads failing return code
                ST    R3,RETURNCODE          Stores return code for   X
                                             future use
EXIT            DS    0H
                L     R13,SAVEBK             Reloads caller's save
*                                            area addr into 11
                L     R12,RETURNCODE         Saves return code
*                                            in reg 12
                LA    R0,DYNSIZE             Loads dynamic area size
*               FREEMAIN R,LV=(0),A=(6),SP=233 Frees dynamic area
                STORAGE RELEASE,SP=233,ADDR=(R6),LENGTH=(R0)
                MODESET KEY=NZERO,MODE=PROB
                LM    R14,R11,12(R13)        Loads return regs
                LR    R15,R12                Loads return code
                BR    R14                    Returns to caller
*
*
         SPACE 2
```

```
*.....................................................................
*
* Subroutine to create Hiperspace
*
*  DSPSERV creates a non-shared standard hiperspace
*
*  Since this sample is for illustration only, the hiperspace size
*  will be one block, or one page, larger than the CROSSOVER value.
*
*.....................................................................
```

```
CREATE_HS     DS       0H               Create Hiperspace routine
              STM      R14,R12,12+LCL_SAVEAREA Save registers
              L        R1,CROSSOVER_#   Obtains the CROSSOVER #
              AL       R1,=F'1'         Adds one to the CROSSOVER
              ST       R1,NUM_BLOCKS    Stores the number of blocks
         DSPSERV CREATE,                                          X
               NAME=HSNAME,                                       X
               TYPE=HIPERSPACE,                                   X
               HSTYPE=SCROLL,                                     X
               SHARE=NO,                                          X
               BLOCKS=NUM_BLOCKS,                                 X
               STOKEN=HSSTOKEN,                                   X
               ORIGIN=HSORIG,                                     X
               MF=(E,DSPSLIST)
              ST       R15,RETURNCODE   Saves return code
              LTR      R15,R15          Test for 0 return code
              BNZ      SKIP_ALESERV     Skip ALESERV if bad RC
*.....................................................................
*
*  The IOSADMF service requires an ALET as input.  The following
*  ALESERV service will place the hiperspace ALET on the program's
*  access list.
*
*.....................................................................
         ALESERV ADD,                                             X
               STOKEN=HSSTOKEN,                                   X
               ALET=HSALET,                                       X
               AL=WORKUNIT,                                       X
               MF=(E,ALESLIST)
SKIP_ALESERV  DS      0H
              LM      R14,R12,12+LCL_SAVEAREA Load Register
              L        R15,RETURNCODE   Loads return code
              BR      R14              Returns to caller
*.....................................................................
*
* Subroutine to obtain and initialize address space storage areas
*
*.....................................................................
AS_STORAGE    DS       0H               Obtains address space storage
              STM      R14,R12,12+LCL_SAVEAREA Save registers
              L        R5,ONE_PAGE
              L        R3,NUM_BLOCKS    Loads the size of hiperspace
*                                       NOTE: For this sample, the
*                                             hiperspace and address
*                                             space areas are made to
*                                             be the same size for
*                                             simplicity.
```

```
              MR       R2,R5            Calculates length of storage
*                                       to obtain
              ST       R3,LENGTH_AS_AREA
              STORAGE OBTAIN,LENGTH=((R3)),BNDRY=PAGE,SP=229
*
              ST      R1,ASPTR          Saves the addr of data area
              L       R2,ASPTR          Loads R2 with address of
*                                        the obtained area in
*                                        preparation for clearing
*                                        using the MVCL.
              L       R3,LENGTH_AS_AREA  Loads length of area into R3
              SR      R4,R4             Setting R4/R5 pair to zero
              SR      R5,R5              tells MVCL to clear area
              MVCL    R2,R4             Clear obtained area
*
              LM      R14,R12,12+LCL_SAVEAREA Load Register
              BR      R14               Returns to caller
*.....................................................................
*
* Subroutine to initialize the address space area with data to be
* stored in the hiperspace.
*
* This subroutine loops through all of the address space buffer
* pages and initializes each page with some text data.
* The data placed in the address space buffer area is dummy data
* for illustration.  It places some text at the top of each
* page and places the page number in hex after the text.
*
*.....................................................................
INIT_AS       DS       0H               Initialize addr space area
              STM      R14,R12,12+LCL_SAVEAREA Save registers
BLOCK_INDEX   EQU      R2               Make loop control easier to   X
                                        read by using equate for index
```

```
AS_POINTER    EQU      R3               Make address space area easierX
                                        to follow by using equate
              LA       BLOCK_INDEX,1    Initializes block index
              L        AS_POINTER,ASPTR Gets addr space pointer
              USING    PAGE_MAP,AS_POINTER  Use the PAGE_MAP dummy     X
                                        section
INIT_LOOP     DS       0H               Beginning of WHILE loop
              CL       BLOCK_INDEX,NUM_BLOCKS  IF block_index greater  X
                                               num_blocks THEN
              BH       INIT_COMPLETE    Exits loop if complete
              MVC      PAGE_TEXT_TAG,BLOCK_CONST     Place text tag
              ST       BLOCK_INDEX,PAGE_INDEX_TAG    Place hex tag
              AL       BLOCK_INDEX,=F'1'             Index to next page
              AL       AS_POINTER,ONE_PAGE           Point to next page
              B        INIT_LOOP
INIT_COMPLETE DS       0H
              LM      R14,R12,12+LCL_SAVEAREA        Load Register
              BR      R14                            Returns to caller
```

```
*.....................................................................
*
* Subroutine to Initialize the Hiperspace
* ---------------------------------------
*
*  HSPSERV initializes hiperspace blocks.  Before the hiperspace
*  can be used by IOSADMF, it must be initialized using the HSPSERV
*  service.  The HSPSERV service causes hiperspace pages to be
*  backed with actual expanded storage pages.  Even though ADMFEXMP
*  created the hiperspace earlier, the system does not actually
*  allocate expanded storage pages until data is placed into them.
*  The following HSPSERV service will cause expanded storage pages
*  to be backed.
*.....................................................................
ISSUE_HSPSERV DS       0H               Initialize hiperspace routine
              STM      R14,R12,12+LCL_SAVEAREA Save registers
              L        R2,ASPTR         Loads address space pointer
              ST       R2,ASPTR1        Saves address space pointer   X
                                         in range list
              L        R2,HSORIG        Loads hiperspace block pointer
              ST       R2,HSORIG1       Saves hiperspace pointer in   X
                                         range list
              L        R2,NUM_BLOCKS    Loads number of blocks to move
              ST       R2,NUMBLKS1      Saves number of blocks to     X
                                         move in range list
              LA       R2,SWRITLST      Loads address of ranglist
              ST       R2,SWRITADDR     Saves address of ranglist
              LA       R13,HSP_SAVEAREA
         HSPSERV SWRITE,                                             X
               STOKEN=HSSTOKEN,                                      X
               HSPALET=HSALET,                                       X
               RANGLIST=SWRITADDR,                                   X
               MF=(E,HSPSLIST)
              ST       R15,RETURNCODE   Saves return code
              LM       R14,R12,12+LCL_SAVEAREA Load Register
              L        R15,RETURNCODE   Loads return code
              BR      R14               Returns to caller
*.....................................................................
*
* Subroutine to use the IOSADMF service to read data from hiperspace
* ------------------------------------------------------------------
*
*  IOSADMF
*
*.....................................................................
ISSUE_IOSADMF DS       0H               Uses the IOSADMF service
              STM      R14,R12,12+LCL_SAVEAREA Save registers
              LR       R3,R1            Loads area addr in R3
              A        R3,LENGTH_AS_AREA  Adds length of area to addr
              BCTR     R3,0             Subtracts 1 to get end addr
              PGSER R,FIX,A=(R1),EA=(R3),ECB=0
              L        R2,ASPTR         Loads R2 with address of      X
                                         the address space area in    X
                                         preparation for clearing     X
                                         using the MVCL.  R3 will     X
                                         contain the area's length
              L        R3,LENGTH_AS_AREA  Loads length to clear
```

```
              SR       R4,R4            Setting R4/R5 pair to zero
              SR       R5,R5             tells MVCL to clear area
              MVCL     R2,R4            Clear target area for the     X
```

```
                                        AREAD operation.  For        X
                                        illustration purposes, the   X
                                        address space area is        X
                                        reused for the ADMF AREAD
             L        R2,ASPTR          Loads address space pointer
             ST       R2,ASPTR2         Saves address space pointer   X
                                         in range list
             L        R2,HSORIG         Loads hiperspace block pointer
             ST       R2,HSORIG2        Saves hiperspace pointer in   X
                                         range list
             L        R2,NUM_BLOCKS     Loads number of blocks to move
             ST       R2,NUMBLKS2       Saves number of blocks to     X
                                         move in range list
             LA       R2,AREADLST       Loads address of ranglist
             ST       R2,AREADADDR      Saves address of ranglist
         IOSADMF AREAD,                                              X
               ALET=HSALET,                                          X
               RANGLIST=AREADADDR,                                   X
               MF=(E,ADMFLIST)
             ST       R15,RETURNCODE    Saves return code
             LM       R14,R12,12+LCL_SAVEAREA Load Register
             L        R15,RETURNCODE    Loads return code
             BR      R14                Returns to caller
*.................................................................
*                                                                 .
*  Constants                                                      .
*                                                                 .
*.................................................................
HSNAME       DC    CL8'ADMFHSPS'            Name for the hiperspace
ONE_PAGE     DC    F'4096'                  Length of one page of
*                                            storage
BLOCK_CONST  DC    CL7'Block #:'
*.................................................................
*                                                                 .
*    DSECTs to map save areas and dynamic areas                   .
*                                                                 .
*.................................................................
DYNSTART     DS    0H
DYNAREA     DSECT
*        Save area
SAVEXX       DS     F
SAVEBK       DS     F
SAVEFW       DS     F
SAVER14      DS     F
SAVER15      DS     F
SAVER0       DS     F
SAVER1       DS     F
             DS   11F
             DS   0D            Force doubleword alignment
*       Save area for internal subroutines
             SPACE 2
LCL_SAVEAREA DS   18F           Local save area
HSP_SAVEAREA DS   32F           HSPSERV save area
             DS   0D            Force doubleword alignment
```

```
*.................................................................
*                                                                 .
*  List forms of macros.  The list and execute forms of these macros  .
*  are used because this module is reentrant.                     .
*                                                                 .
*.................................................................
LIST_DSPSERV DSPSERV MF=(L,DSPSLIST)
DSP_END      DS   0D
LIST_HSPSERV HSPSERV MF=(L,HSPSLIST)
HSP_END      DS   0D
LIST_IOSADMF IOSADMF MF=(L,ADMFLIST)
ADMF_END     DS   0D
ALESLIST     ALESERV MF=L
ALES_END     DS   0D
*.................................................................
*                                                                 .
*  Work variables and data structures local to this module        .
*                                                                 .
*.................................................................
HSSTOKEN     DS    CL8                  STOKEN for the hiperspace
HSALET       DS    CL4                  ALET for the hiperspace
ASPTR        DS    1F                   Location of addr space
*                                         storage
NUM_BLOCKS   DS    F                    Number of blocks in
*                                         hiperspace
HSORIG       DS    F                    Hiperspace origin
```

```
CROSSOVER_#   DS    F                       Crossover number
SWRITADDR     DS    F                       Address of SWRITE ranglist
AREADADDR     DS    F                       Address of AREAD ranglist
ADMF_INSTALLED_RC DS F                      ADMF installed return code
LENGTH_AS_AREA DS   F                       Length of addr space area
WORKAREA      DS    CL144                   Work area for HSPSERV
               DS    0F
SWRITLST      DS    0CL12                   SWRITE range list
ASPTR1        DS    F                       Start of address space
HSORIG1       DS    F                       Target location in hiperspace
NUMBLKS1      DS    F                       Number of 4k blks in swrite
               DS    0F
AREADLST      DS    0CL12                   AREAD and SREAD range list
ASPTR2        DS    F                       Target location in AS
HSORIG2       DS    F                       Start of hiperspace source
NUMBLKS2      DS    F                       Number of 4k blocks in read
*
RETURNCODE    DS    F
REASONCODE    DS    F
END_DYN       DS    0D
DYNSIZE       EQU   *-DYNAREA                Calculates Dynamic area
*
```

```
PAGE_DSECT   DSECT                          Mapping of a page
PAGE_MAP      DS    0CL4096
PAGE_TEXT_TAG DS    CL8                     Top of page tag
PAGE_INDEX_TAG DS   F                       Page index in hex
         SPACE 2
ADMFEXMP      CSECT
              TITLE 'ADMFEXMP - DSECT MAPPINGS'
         EJECT
         CVT   LIST=YES,DSECT=YES
         END   ADMFEXMP
```

Extending the current size of a hiperspace

When you create a hiperspace and specify an initial size smaller than the maximum 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 system increases the hiperspace by the amount you specify, unless that amount would cause the system to exceed one of the following:

- The hiperspace maximum size, as specified by the BLOCKS parameter on DSPSERV CREATE when the hiperspace 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 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 hiperspace as much as possible without exceeding the limits set by the hiperspace maximum size or the installation limits. In other words, the system extends the hiperspace 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
  - Reject the request, if the hiperspace has storage key 8 - F and the request would exceed the installation limits.

For example, consider the hiperspace in Figure 51 on page 167, where the current (and initial) size is 20,000 bytes and the maximum size is 100,000 bytes. If the creating program wanted to increase the current size to 50,000 bytes by adding a 30,000 bytes to the current size, it would code the following:

```
          DSPSERV EXTEND,STOKEN=HSSTOK,BLOCKS=HSBLCKS
          .
HSDELTA EQU  30000                  30000 BYTES OF SPACE
HSBLCKS DC   A((HSDELTA+4095)/4096) NUMBER OF BLOCKS ADDED TO THE
*                                   HIPERSPACE
HSSTOK  DS   CL8                    STOKEN RETURNED FROM DSPSERV CREATE
```

The storage the program can use in the 100,000 byte hiperspace would then be the first 50,000 bytes, as shown in Figure 61 on page 191:

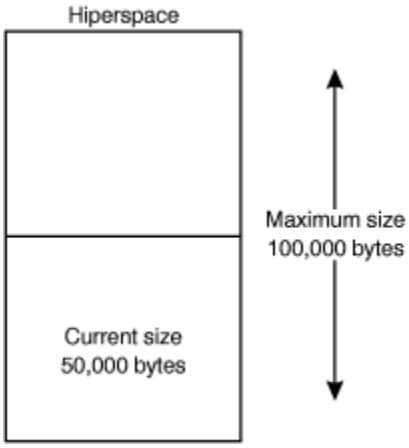


Figure 61. Example of extending the current size of a hiperspace

If you use VAR=YES when you issue the EXTEND request, use NUMBLKS to find out the size by which the system extended the hiperspace.

Deleting a hiperspace

When a program doesn't need the hiperspace any more, it can delete it. A problem state program with PSW key 8 - F can delete only the hiperspaces it owns, and must have the PSW key that matches the storage key of the hiperspace. A supervisor state program or a program with PSW 0 - 7 can delete a hiperspace if its home or primary address space is the same as the owner's home address space.

If you are not the owner of a hiperspace you are using, the hiperspace might disappear if the owner terminates or deletes it. For example, a problem state program can delete a hiperspace that a supervisor state program is using.

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. This frees up resources as soon as they are no longer needed, and avoids excess processing at termination time. However, if you don't, MVS automatically does it for you at termination time.

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 (that is, initialize to hexadecimal zeroes and return the resources to the system) the virtual storage of a hiperspace, use the

DSPSERV RELEASE macro. 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 is subject to the following conditions. If these conditions are not met, the system abnormally ends the caller.

- If the hiperspace is a shared standard type or an ESO type
  - The owner must be authorized (supervisor state or PSW key 0-7).
  - The caller's PSW key must be zero or equal to the key of the hiperspace storage the system is to release.
- If the hiperspace is a non-shared standard type and the caller is not authorized
  - The owner's home address space must be the same as the caller's home address space.
  - The caller's PSW key must be equal to the key of the hiperspace storage the system is to release.
- If the hiperspace is a non-shared standard type and the caller is authorized, the caller's PSW key must be zero or equal to the key of the hiperspace storage the system is to release.

After the release, the released pages do not use expanded (or auxiliary) storage until your program references them again. When such a page is referenced again, these pages contain hexadecimal zeroes.

Pages released through DSPSERV RELEASE do not occupy space in expanded or auxiliary storage. The pages are available for you to use, and they contain hexadecimal zeroes.

If your program is running disabled for I/O or external interrupts, use the DISABLED=YES parameter on DSPSERV RELEASE. If your program is running disabled and issues DSPSERV RELEASE without DISABLED=YES, the system abends the program.

Using data-in-virtual with standard 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 as Table 21 on page 192 describes:

Table 21. Uses of hiperspaces and data-in-virtual

Question about behavior	Non-shared standard hiperspace	Shared standard hiperspace	ESO hiperspace
Can the hiperspace map a VSAM linear data set?	Yes	Yes	No
Can the hiperspace be a data-in-virtual object?	Yes, providing the hiperspace has not been the target of an ALESERV ADD	No	No

The task that represents the program 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 a problem state program with PSW key 8 cannot issue the DIV IDENTIFY and another problem state program with PSW key 8 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 non-shared or shared standard hiperspace, providing the program owns the hiperspace.

- Map an object in a non-shared hiperspace to an address space window, providing:
  - The program owns the hiperspace, and
  - The program or its attaching task obtained the storage for the window (through the STORAGE or GETMAIN macro), and
  - The hiperspace has never been the target of an ALESERV ADD macro.

Data-in-virtual allows supervisor state programs or programs with PSW key 0 - 7 (called "authorized programs" in this section) to issue DIV IDENTIFY and then have subtasks of that task use the structures. The subtasks must also be authorized. This means that an authorized program can issue a DIV IDENTIFY and an authorized subtask can issue the DIV MAP for that ID.

Table 22 on page 193 shows what data-in-virtual requires of the tasks that represent the programs that issue the DIV macros. The table does not show the IDENTIFY service because data-in-virtual does not have restrictions on this service.

Table 22. Requirements for authorized programs using the DIV services with hiperspaces

Question about behavior	ACCESS	MAP	SAVE	UNIDENTIFY, UNACCESS, UNMAP, RESET
Object is a linear data set, window is in a non-shared or shared standard hiperspace	Task that issued the DIV IDENTIFY.	Task (or authorized subtask of the task) that issued the DIV IDENTIFY. (See Note 1.)	Task (or authorized subtask of the task) that issued the DIV IDENTIFY. The task does not have to own the hiperspace.	Task (or authorized subtask of the task) that issued the DIV IDENTIFY. The task does not have to own the hiperspace.
Object is a non-shared standard hiperspace, window is in an address space	Task that issued the DIV IDENTIFY. The task must own the hiperspace.	Task that issued the DIV IDENTIFY. The task (or a supertask of the task) that issued the DIV IDENTIFY must have obtained storage for the window. (See Note 2.)	Task that issued the DIV IDENTIFY.	Task (or authorized subtask of the task) that issued the DIV IDENTIFY. The task does not have to own the hiperspace.

Note:

1. If the program is in supervisor state or PSW key 0 - 7, any task within the caller's primary address space can own the hiperspace.
2. If the program is APF-authorized, but not supervisor state or PSW key 0 - 7, the caller must own the hiperspace.
3. A task cannot map to virtual storage that a subtask obtained. However, a super task (that is, a task higher in the TCB chain) can obtain the storage.

Whether the hiperspace contains the window or is the object, the data-in-virtual service will not create a local copy of the object (that is, you cannot use the LOCVIEW=MAP parameter on DIV ACCESS).

The following two sections describe how your program can use data-in-virtual 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. It is recommended that you obtain the ALET for the hiperspace and use the HSPALET parameter on HSPSERV to get faster data transfer to and from expanded storage.

When a program maps a data-in-virtual object to a non-shared or shared 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 the 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 non-shared standard hiperspace with a maximum size of one gigabyte and an initial size of 4K bytes. Figure 62 on page 194 shows the hiperspace with a window that begins at the origin of the hiperspace.

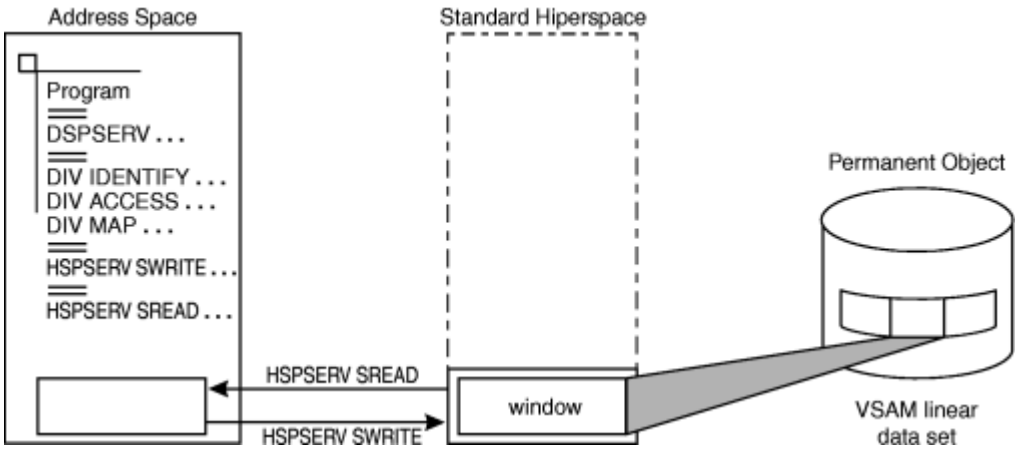


Figure 62. Example of mapping a data-in-virtual object to a hiperspace

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 use the HSPALET parameter to gain faster access to and from expanded storage. The HSPSERV SWRITE macro can update the data and write changes back to the hiperspace. For an example of these operations, see “Example of creating a standard hiperspace and using It” on page 178.

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 no program has ever issued an ALESERV ADD for the hiperspace. With the hiperspace as the object, 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 63 on page 195 shows an example of a hiperspace as a data-in-virtual object.

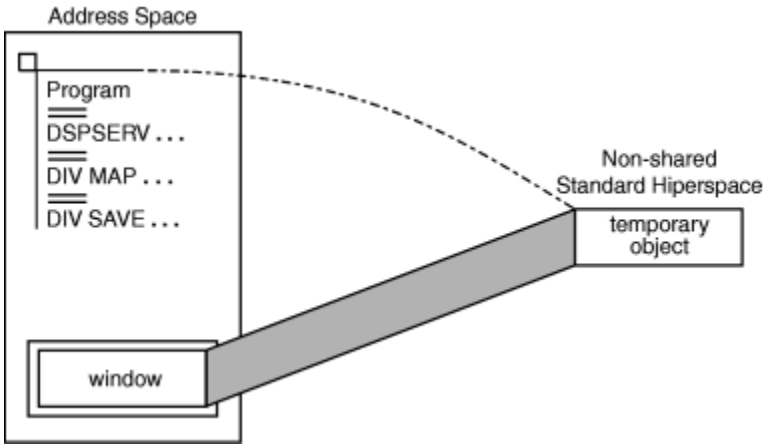


Figure 63. A Standard hiperspace as a data-in-virtual object

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 cannot use the HSPSERV read and write operation.

An example of a hiperspace as a data-in-virtual object

The program in this section creates a non-shared standard 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 non-shared 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 DS   F'1'                HIPERSPACE SIZE OF 1 BLOCK
OBJID    DS   CL8                 DIV OBJECT ID
OBJAREA  DC   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 have 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
```

How SRBs use hiperspaces

An SRB cannot own a hiperspace. Through the DSPSERV CREATE macro, a program in supervisor state or PSW key 0 - 7 can assign ownership of a hiperspace to a TCB. The owning TCB must reside in the SRB's home or primary address space.

When you use the DSPSERV CREATE macro to create the hiperspace and assign ownership, you must identify the TCB through the TTOKEN parameter. A TTOKEN identifies a TCB. Unlike TCB addresses, TTOKENs are unique within the IPL; the system does not assign this same TTOKEN to any other TCB until the next IPL. If you know the TCB address, but not the TTOKEN for the task that is to receive ownership, use the TCBTOKEN macro. The TCBTOKEN macro accepts the TCB address and returns a TTOKEN. You then use this TTOKEN in the DSPSERV CREATE macro. For more information about TTOKENs, see “Obtaining the TCB identifier for a task (ttoken)” on page 154.

When an SRB terminates, it can delete any hiperspaces it created. Use the TTOKEN parameter on the DSPSERV DELETE macro to specify the address of the TTOKEN of the hiperspace owner.

Chapter 8. Creating address spaces

This section is for the programmer who wants to create an address space. Perhaps the address space is for a subsystem that has "outgrown" the address space it shared with other programs; perhaps it will be for a set of programs that provide a service for programs in other address spaces. Perhaps the subsystem or the programs need to have more control over their environment.

One way for a program to create an address space, without involving an MVS operator, is by issuing a START command through the MGCR macro. The program must have a procedure in SYS1.PROCLIB, representing the first program that will execute in the created address space. The program can assign the dispatching priority for the programs that will run in the created address space. The initialization can include cross memory macros that establish a cross memory environment for the created address space.

An easier way to accomplish the same objectives is to issue the **ASCRE macro.** The ASCRE macro creates a address space that can start after the system is initialized and receive the services of all MVS components. It can set up cross memory linkages so that programs in the created address space can call programs in the creating program's address space. It can set a dispatching priority for the programs that run in the created address space and can specify that the address space exist after the task that represents the creating program terminates. (For simplicity, in this section the term "creating program" refers to the address space of the program that is issuing the ASCRE macro. The term "new address space" refers to the address space that the ASCRE macro creates.)

The system considers the new address space to be a system address space, and it will show up as such when the operator issues a DISPLAY A command, unless ATTR=JOBSPACE is specified on the ASCRES macro. Program properties table (PPT) values determine the attributes of the programs that execute in the new address space.

This section describes how to use the ASCRE macro and two other macros that assist in managing address spaces:

- The **ASDES macro** terminates an address space that was created through the ASCRE macro.
- The **ASEXT macro** retrieves parameters that the creating program passes to a program in the new address space.

For the syntax of the macros mentioned in this section, see one of the following:

- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

Using the ASCRE macro to create an address space

The ASCRE macro creates an address space. Parameters on the macro allow you to:

- Name the address space and specify the first program that is to run in the address space (ASNAME or STPARM parameter).
- Specify an address space initialization routine that ASCRE processing calls (INIT parameter).
- Provide a routine to control the termination of the address space (TRMEXIT parameter).
  - Provide the termination routine with user data (UTOKEN parameter).
- Pass a parameter list from the creating program to a program in the new address space (ASPARM or STPARM parameter).
- Assign various attribute to the new address space (ATTR parameter).
- Set up cross memory linkages between the creating address space and the new address space (AXLIST, TKLIST, and LXLIST parameters).

- Provide an address where ASCRE will return output data (ODA parameter).

The required parameters on the ASCRE macro are ASNAME or STPARM, INIT, and ODA. Some comments about these parameters are:

- Specify either ASNAME or STPARM.
  - ASNAME specifies the address space name, which is also the name of the procedure that identifies the first program to run in the new address space.
  - STPARM specifies a parameter string. Although the system does not actually issue a START command, the parameter string consists of START command parameters. This parameter string must begin with the name of the address space.

  An operator can use the name of the address space on the DISPLAY A command to display information about the address space. For the syntax of the DISPLAY command, see *z/OS MVS System Commands*.
- Output data from the macro appears at the location specified on the ODA parameter. This data includes two identifiers of the new address space: the ASCB and the STOKEN. The STOKEN is an identifier that is unique within the lifetime of an IPL. The format of the output data area is a 24-byte area that the macro IHAASEO maps as follows:

Offset	Length	Description
X‘00’	8 bytes	The STOKEN of the new address space
X‘08’	4 bytes	The ASCB of the new address space
X‘0C’	4 bytes	Basing for IEZEAECB, the mapping macro for the two ECBs, EAERIMWT and EAEASWT
X‘10’	8 bytes	Reserved

  For the format of IHAASEO, see the ASEO data area in *z/OS MVS Data Areas* in the z/OS Internet library (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary).
- Although INIT is a required parameter, you might not need to write an initialization routine. “The address space initialization routine” on page 200 can help you decide whether you need an initialization routine.

Issue the ASCRE macro in a supervisor state program that is executing in primary or AR address space control (ASC) mode. The caller must be in enabled and unlocked TCB mode and must not have an enabled unlocked task (EUT) functional recovery routine (FRR) established. Issue the ASCRE macro only after system initialization is complete.

Planning the characteristics of the address space

Before you issue the ASCRE macro, you have some planning to do and some actions to take. For a list of the questions you have to consider before you issue the macro, look at Table 23 on page 198. This figure lists the questions, describes the actions that you might take, and points you to the section in this section where you can find information about the action.

Table 23. Planning considerations for the new address space

Considerations	Actions that Might Follow	Reference
Which parameter identifies the procedure in SYS1.PROCLIB that identifies the first program to run in the new address space?	Specify either ASNAME or STPARM on the ASCRE macro.	“Identifying a procedure in SYS1.PROCLIB” on page 199
Do any data sets need special DD statements?	Supply the DD statements in the SYS1.PROCLIB member.	“Identifying a procedure in SYS1.PROCLIB” on page 199

Table 23. Planning considerations for the new address space (continued)		
Considerations	**Actions that Might Follow**	**Reference**
Does the address space need an initialization routine?	Write an initialization routine and specify it on the INIT parameter.	"The address space initialization routine" on page 200
Will the new address space be able to provide cross memory services for programs in other address spaces?	Code the cross memory macros in the initialization routine.	"The new address space as service provider" on page 202
Will programs in the new address space be able to call programs in the creating program's address space?	Code the TKLIST, LXLIST, and AXLIST parameters on the ASCRE macro.	"The new address space as cross memory user" on page 202
Does the address space need a termination routine?	Write a termination routine and code the TRMEXIT and UTOKEN parameters.	"Providing an address space termination routine" on page 205
How will the system dispatch programs in the new address space?	Code NONURG or HIPRI on the ATTR parameter on the ASCRE macro.	"Establishing attributes for address spaces" on page 206
Will the address space exist after the creating task terminates?	Code PERM on the ATTR parameter on the ASCRE macro.	"Establishing attributes for address spaces" on page 206

Identifying a procedure in SYS1.PROCLIB

The procedure in SYS1.PROCLIB specifies the first program to run in the new address space after the optional initialization routine. You can specify the name of this procedure either on the ASNAME or STPARM parameter.

- On the ASNAME parameter, you specify the name of the new address space, which must be the same as the name of the procedure in SYS1.PROCLIB. (This parameter assumes that you have the procedure in SYS1.PROCLIB.) You cannot pass parameters to JCL through ASNAME.
- On the STPARM parameter, you specify the address of a parameter string that consists of a two-byte length field followed by up to 124 bytes of parameter data. The length field identifies the length of the parameter data (not including the length field itself). The parameter data begins with the name of the new address space, which must be the same as the name of the procedure in SYS1.PROCLIB. The name is followed by parameters. You can pass parameters to JCL through STPARM.

If you do not need special DD statements for data sets, you can use the common system address space procedure IEESYSAS. In the parameter data specified by the STPARM parameter, specify IEESYSAS and the name of the first program to run in the address space. The format of the parameter data is as follows:

```
IEESYSAS.x,PROG=y
```

where:

x
The name of the address space

y
The name of the first program that executes in the new address space

Through IEESYSAS, you name the address space "x" and generate an EXEC statement with PGM="y".

Example of a parameter string

To request that the system create the RMA address space and identify FIRSTPGM as the first program to execute in the new address space, code the following:

```
ASCRE STPARM=STRMA,...
```

where STRMA is the address of the parameter string.

The parameter string is coded as follows:

```
STRMA  DS    0H
       DC    H'26'
       DC    CL26'IEESYSAS.RMA,PROG=FIRSTPGM'
```

where:

H'26'
Indicates that the parameter string is 26 characters long

IEESYSAS
Identifies the procedure to be used

RMA
The name of the new address space

FIRSTPGM
The name of the first program in the new address space

If you have data sets that need DD statements, you will have to write your own procedure in SYS1.PROCLIB. Identify the procedure through the parameter string that the STPARM parameter points to. The parameter string starts with a halfword field that tells the length of the parameter string. It is followed by parameter data.

The address space initialization routine

The initialization routine is a program that you can write to set up certain services or data areas for the new address space. It executes in the new address space before the procedure identified by SYS1.PROCLIB. Each initialization routine initializes an address space according to unique requirements of the address space. For example, the initialization routine might:

- Create and initialize control blocks
- Load executable code
- Build the cross memory linkages to allow programs in other address spaces to call PC routines in the new address space.

The AXLIST, TKLIST, and LXLIST parameters on ASCRE set up cross memory linkages that allow programs in the new address space to be cross memory users, but not cross memory service providers. The initialization routine is a good place to use the cross memory macros that allow PC routines in the new address space to be invoked from other address spaces.

The address space that ASCRE is to create might not need initializing beyond what the macro provides. In this case, specify the dummy routine IEFBR14 on the INIT parameter and ignore the following description of the initialization routine.

Writing an Initialization Routine

The system program invokes the initialization routine in supervisor state. The routine must reside in the link pack area (PLPA, MLPA, or fixed LPA) or in a library in the LNKLST concatenation.

On entry to the initialization routine:

- Register 1 contains the address of the parameter list, which contains the following:
  - Address of the newly created address space's ASCB (mapped by IHAASCB)
  - Address of ECBs (mapped by IEZEAECB)
- Register 13 contains the address of a standard 18-word save area.
- Register 14 contains the return address.

- Register 15 contains the address of the initialization routine.

Synchronizing the Initialization Process

The caller of ASCRE might want to wait until the new address space is initialized (the point at which the initialization routine has finished processing and has returned to the system program) before starting to run the first program in the new address space. The system provides the caller with two ECBs — EAERIMWT and EAEASWT — that it can use for communication and synchronization between the creating program and the initialization routine. The address of these two ECBs is contained in the data area that the ODA parameter specifies. The format of the 24-byte output area appears earlier in this section, and the macro IEZEAECB maps the two ECBs within that data area. For the format of the ECBs, see the EAECB data area in *z/OS MVS Data Areas* in the z/OS Internet library (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary).

Use the ECBs as follows:

- When the initialization routine gets control and completes all or part of its processing, it posts EAERIMWT (a "cross memory" POST) to notify the caller of ASCRE. If the initialization routine needs further communication with the creating program, it can wait on EAEASWT. The initialization routine can pass up to three bytes of information in the post code of the ECB.

  Note: To post EAERIMWT with a cross memory post, the initialization routine must supply the creating address space's ASCB address on the POST macro. The routine obtains the ASCB address from the creating program in the following manner:

  – The creating program obtains the ASCB address from PSAAOLD
  – The creating program places the ASCB address in a parameter string, and specifies the parameter string on the ASPARM parameter on the ASCRE macro
  – The initialization routine invokes the ASEXT macro to obtain the address of a copy of the parameter string.
- If the caller needs to communicate back to the initialization routine, it posts EAEASWT, which the initialization routine is waiting on.

Figure 64 on page 201 shows an example of a program creating a new address space that has PC services. The caller of ASCRE does not want PC routines in the new address space to be called from other address spaces before the cross memory environment is initialized and able to handle those requests.

IBM recommends that your initialization routine post EAERIMWT and wait on EAEASWT. If the initialization routine posts EAERIMWT, but does not wait on EAEASWT, the system frees the ECBs prematurely. This action causes the system to abend the caller of ASCRE.

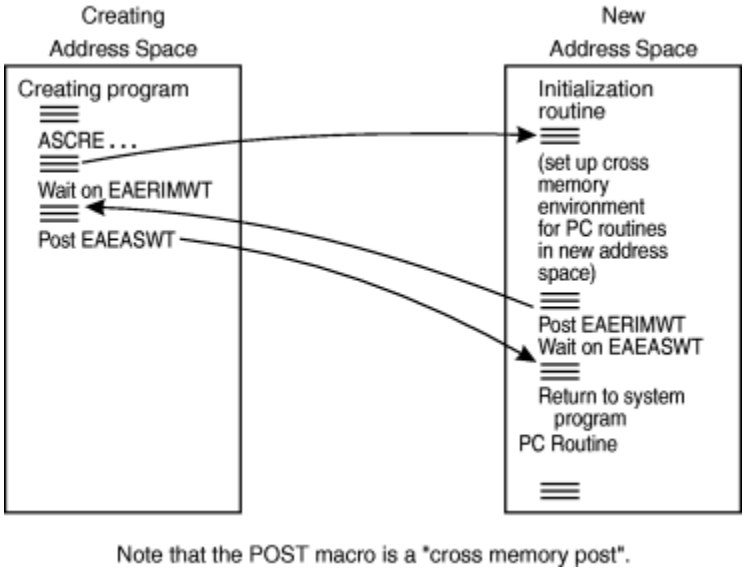


Figure 64. Synchronization of the address space creation process

The initialization routine should use one of the following reason codes when it returns to the system program:

- 0 — Continue with address space initialization
- 4 — Terminate the address space.

If the initialization routine is going to return to the system program with the return code that requests that the new address space terminate, the routine should first notify the caller of ASCRE to allow the system to continue processing. To notify the caller:

- The initialization routine can post EAERIMWT and wait on EAEASWT.
- When the caller is posted, it can post EAEASWT to notify the initialization routine. Control returns to the system program.

Establishing cross memory linkages

There are two types of cross memory environments that you can set up between the creating address space and the new address space. In one, programs in the new address space provide services for other address spaces. In the second, programs in the new address space use services provided by other address spaces. Use this section along with Chapter 3, “Synchronous cross memory communication,” on page 19 to set up either environment.

The new address space as service provider

Figure 65 on page 202 illustrates the environment in which the new address space will be a service provider.

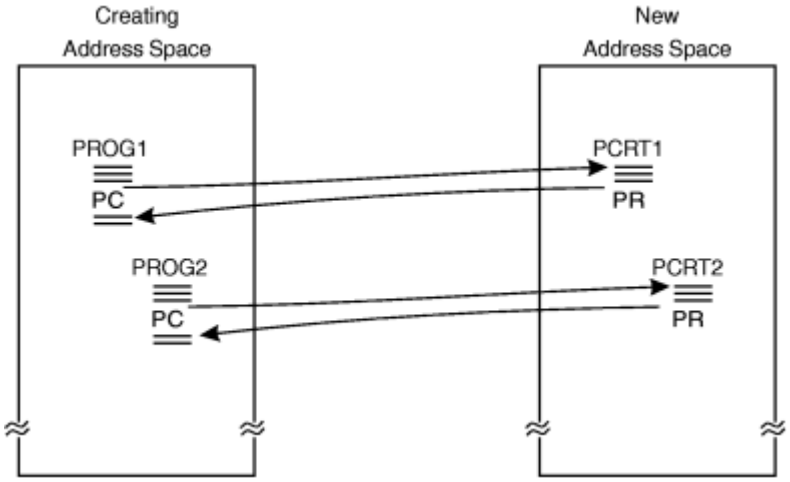


Figure 65. An example of a cross memory environment

In this figure, programs PROG1 and PROG2 in the creating address space can use the PC services PCRT1 and PCRT2 provided by the new address space.

To set up the cross memory linkages for this environment in Figure 65 on page 202, cross memory macros must be issued in the new address space. The initialization routine, which runs in the new address space, is a good place to set up this environment. For help in writing this initialization routine, see “Writing an Initialization Routine” on page 200. Figure 64 on page 201 shows how you synchronize the initialization process so that PC routines in the new address space are not called before the cross memory linkages are in place.

The new address space as cross memory user

Parameters on the ASCRE macro enable the creating program to establish the second type of cross memory environment — one in which cross memory linkages enable the new address space to use the PC services provided by the creating program's address space. The environment is in place as soon as ASCRE processing is complete.

The cross memory environment that ASCRE can set up is described in Figure 66 on page 203. Programs in the new address space (PGM1 and PGM2) can call PC routines (PCRTN1 and PCRTN2) in the creating program's address space. (ASCRE cannot set up the environment where programs in the creating program's address space can call PC routines in the new address space.)

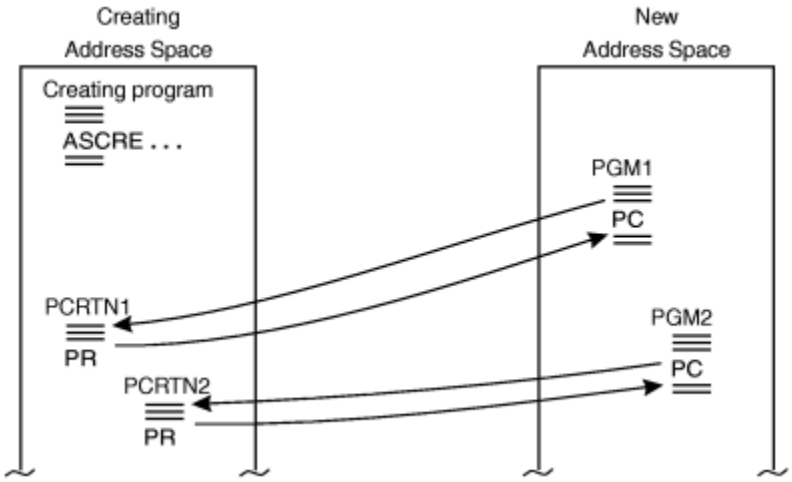


Figure 66. An example of cross memory environment set by the ASCRE macro

There are three cross memory parameters on ASCRE. TKLIST and LXLIST can be used to connect an entry table (or tables) of the creating program's address space to the linkage table of the new address space. AXLIST can be used to set the values in the authority table of the new address space so that programs in the creating program's address space have PT and SSAR authority to the new address space. The effect of the cross memory parameters is the same as if the new address space issued the ATSET and ETCON macros, macros that require a program to be in supervisor state or key 0 - 7.

The three cross memory parameters require that you provide the addresses of three lists. The lists are:

- The AX value (or values) of the creating program's address space
- The linkage index (LX) value (or values) of the new address space's linkage table
- The token (or tokens) that represents the entry table (or tables) in the creating program's address space.

The output of the AXRES and LXRES macros provides the input to the AXLIST and LXLIST parameters. You must build the list that is input to the TKLIST parameter. The following description of the parameters includes the format of the list you provide the TKLIST parameter.

- The **AXLIST parameter** identifies the address of a list of authorization index (AX) values that represent the address spaces that have access to the new address space. The AXRES macro returned these values.

  The effect of using this parameter is the same as if a program in the new address space were to issue the ATSET macro once for every AX value in the list.
- The **LXLIST parameter** identifies the address of a list of linkage index (LX) values that represent entries in the new address space's linkage table. The LXRES macro returned these values. The number of linkage indexes for LXLIST must be the same as the number of tokens for TKLIST.

  The effect of using this parameter is the same as if a program in the new address space issued the ETCON macro with the LXLIST parameter.
- The **TKLIST parameter** identifies the address of a list of fullword tokens — each token represents an entry table to which the system will connect the new address space's linkage table. You build this list with tokens that the ETCRE macro returned, one token for each occurrence of the ETCRE macro. Format the list as follows:
  - The first fullword contains the number of tokens in the list
  - Up to 32 fullwords follow, each containing one entry table token.

The effect of using this parameter is the same as if a program in the new address space issued the ETCON macro with the TKLIST parameter.

If the creating address space is to be a service provider for the new address space, the creating program (or another program in the same address space) must have issued the following cross memory macros before it issues the ASCRE macro:

- AXRES macro, which reserves an AX value (or values)
- AXSET macro, which sets an AX
- LXRES macro, which reserves an LX value (or values)
- ETCRE macro, which creates an entry table and returns a token that identifies the table.

On the ASCRE macro, the creating program can:

- Use the AXLIST, LXLIST, and TKLIST parameters to set up a cross memory linkage with the new address space.
- Use the ASPARM parameter to pass the PC number of a PC routine in the creating program's address space.

  A program in the new address space can later use the ASPARM parameter on the ASEXT macro to obtain a PC number or numbers. See “Passing a parameter list to the new address space” on page 205 for more information about passing parameters to the new address space.

Figure 67 on page 205 shows the same cross memory environment that Figure 66 on page 203 showed. The creating address space would give the following input to ASCRE:

- As input on AXLIST, the address of a list containing the AX value of the creating address space. The AXRES macro returned the address.
- As input on TKLIST, the address of a list that you created. The first entry in the list is X‘0001’, the second entry is the entry table token that the ETCRE macro returned.
- As input on LXLIST, the address of a list containing the LX value of the new address space's linkage table. The LXRES macro returned the address.

The AXLIST parameter sets the authority table in the new address space so that PC routines in the creating address space can have address space authorization to the new address space. TKLIST and LXLIST connect the entry table in the creating address space to the new address space's linkage table.

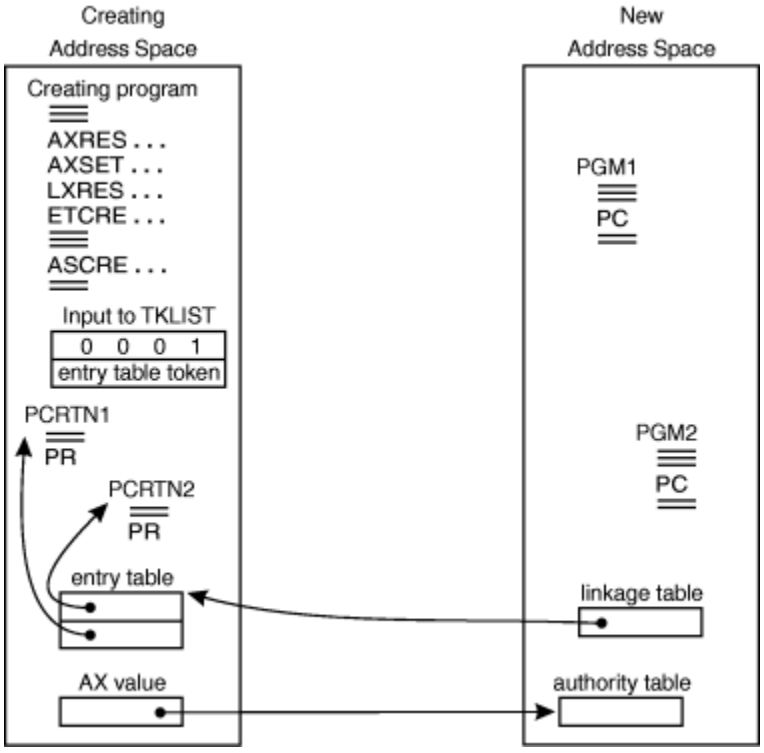


Figure 67. The cross memory linkages set by the ASCRE macro

The AXLIST parameter is not always needed to set up the cross memory linkages. If the address space to which the creating program needs access has only one AX value and that value is 1, the system does not need to initialize entries in the new address space's authority table. (An AX of 1 is a fully-authorized AX value that permits the address space to issue PT and SSAR instructions to any active address space.) If all AX values are 1, you can omit the AXLIST parameter.

Passing a parameter list to the new address space

Through the ASCRE macro, you can pass up to 254 bytes of data in a parameter string to a program in the new address space. The ASPARM parameter identifies the address of the parameter string. A program in the new address space can use the ASEXT macro to obtain a copy of that parameter string. The ASEXT macro returns the address of the copy in register 1. "Example of creating and deleting an address space" on page 207 contains an example of a program in the new address space retrieving a copy of the parameter string.

Providing an address space termination routine

You can provide a routine that receives control when the address space terminates for reasons other than the ASDES macro. Specify the address of the routine on the TRMEXIT parameter. The termination routine receives control in 31-bit addressing mode, supervisor state, and in the current TCB key of the task that issued the ASCRE macro. The routine is an asynchronous exit (IRB) that resides in the address space of the creating program and runs under its TCB.

To identify the terminating address space to the termination routine and pass data, use the 64-bit area provided on the UTOKEN parameter. Specify the address of this data area on the UTOKEN parameter.

The UTOKEN parameter provides a 64-bit area in which the creating program can pass data to the termination routine. Use UTOKEN to:

- Give the address space a unique name that the termination routine can use. If your program has created more than one address space, the UTOKEN parameter area can identify which address space is terminating.

- Pass other data to the termination routine. Data might consist of the address and ALET of a workarea containing the name of the address space.

On entry to the routine:

- Register 1 contains the address of a copy of the 64-bit data that the UTOKEN parameter supplies.
- Register 13 conditionally contains the address of a standard 18-word save area.

  A save area is provided only in the following cases:

  – The TCB key of the caller of the ASCRE macro is 0 - 7.
  – The job step of the caller is APF-authorized and is not running the TSO TMP.

  The save area is obtained in virtual storage that is normally assigned the storage key of the TCB. Note that the system does not require the exit routine to save and restore registers. For exiting purposes, the exit routine can return using the address in input register 14 or by branching to CVTEXIT.
- Register 13 conditionally contains the address of a standard 18-word save area, providing either of the following conditions are met:

  – The TCB key of the caller of the ASCRE macro is 0 - 7.
  – The job step of the caller is APF-authorized and is not running the TSO/E TMP.

  The save area is obtained in virtual storage that is normally assigned the storage key of the TCB. Note that the system does not require the exit routine to save and restore registers. For exiting purposes, the exit routine can return using the address in input register 14 or by branching to CVTEXIT.
- Register 14 contains the return address.
- Register 15 contains the entry point address.

The termination routine will not get control for either of the following two reasons:

- The ASDES macro terminates the new address space.
- The creating address space terminates before the new address space terminates.

Establishing attributes for address spaces

Through the ATTR parameter, you can define some of the attributes for programs that run in the new address space. Table 24 on page 206 describes the options for ATTR and identifies what definitions override those options.

Table 24. ATTR options for address spaces

Option	Description	Override
NONURG	The address space is to be used by nonurgent services. The defaults for dispatch priority as well as service class or performance group depend on whether the address space is privileged or not. The dispatching priority of programs that run in the new address space is to be low.	The service definitions of the workload management (WLM) policy
HIPRI	The address space is to be used by services with a high priority. The defaults for dispatch priority as well as service class or performance group depend on whether the address space is privileged or not. The dispatching priority of programs that run in the new address space is high.	The service definitions of the WLM policy
PERM	The address space does not terminate when the task that created the address space terminates.	No override

Table 24. ATTR options for address spaces (continued)		
Option	**Description**	**Override**
JOBSPACE	The address space should be marked as a job (started task) space, rather than as a system address space.	No override

The NONURG and HIPRI parameters set the dispatching priority for programs that execute in the new address space. Your installation can also set the dispatching priority through service definitions in the WLM policy. See *z/OS MVS Planning: Workload Management*.

Do not set the dispatching priority in more than one way.

If you want the new address space to exist after the TCB of the creating program terminates, use ATTR=(PERM).

When the address space is marked as a job, the resource name used for the started class profile will be procname JOBNAME. Display it with the DISPLAY A,L command.

Deleting an address space

Use the ASDES macro to delete an address space that was created through the ASCRE macro. The STOKEN parameter is required on the ASDES macro. You can obtain this value from the data output field located at the address specified on the ODA parameter. The macro IHAASEO maps the 24-byte data area specified by ODA. For the format of IHAASEO, see the ASEO data area in *z/OS MVS Data Areas* in the z/OS Internet library (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary).

Note that the termination routine that was optionally specified on the TRMEXIT parameter on the ASCRE macro does not get control in this case.

The ASDES process is similar to what CALLRTM TYPE=MEMTERM provides. Be aware that tasks in the abending address space cannot perform recovery and task-level resource managers cannot get control; address space recovery routines and resource managers can get control. Instead of using ASDES, you might use CALLRTM TYPE=ABTERM,RETRY=NO to abend each job step task in the address space. When all tasks in the address space have terminated, the system deletes the address space.

Example of creating and deleting an address space

The following supervisor state, key 0 program (CREATOR) creates an address space and a cross memory environment in which programs in the new address space (NEWADS) can PC to the creating address space. In this example, CREATOR builds a linkage table (LT) and a corresponding entry table (ET) that allows programs in NEWADS to PC to the routine PCTARGET, which is loaded in the creating address space. CREATOR passes the PC number needed to get to PCTARGET through the ASPARM field on the ASCRE macro. A program in NEWADS can then use the ASEXT macro to extract ASPARM. CREATOR also builds NEWADS's authority table (AT) so that it can PT and SSAR to NEWADS. Note that NEWADS is the name of a procedure (PROC) that resides in SYS1.PROCLIB. This PROC contains the JCL that identifies the first program to run in NEWADS.

```
CREEXMPL CSECT
CREEXMPL AMODE 31
CREEXMPL RMODE ANY
         BAKR  14,0                  SAVE CALLER'S STATUS ON STACK
         BASR  12,0                  SET BASE REGISTER GR
         USING *,12
         .
```

```
         .
*RESERVE AN LX
         LA R4,1
         ST R4,LXCOUNT               NEED ONLY ONE LX
         LXRES LXLIST=LX_LIST        RESERVE AN LX
        .
```

```
*RESERVE AN AX AND SET THE CURRENT ADDRESS SPACE'S AX TO IT
         LA R4,1
         STH R4,AXENTRIES             NEED ONLY ONE AX
         AXRES AXLIST=AX_LIST         RESERVE AN AX
         AXSET AX=AXENTRY             SET HOME'S AX TO THE RESERVED AX
         .
```

```
*DEFINE AN ENTRY TABLE ENTRY FOR THE ROUTINE PCTARGET
ETELIST  ETDEF TYPE=INITIAL           START AN ET ENTRY LIST
         ETDEF TYPE=ENTRY,ROUTINE=PCTARGET,AKM=0,EK=0,EKM=0,PKM=OR,    X
               PC=STACKING,SASN=OLD,SSWITCH=YES,STATE=SUPERVISOR
         ETDEF TYPE=FINAL       COMPLETED THE ET ENTRY LIST
         .
*CREATE AN ENTRY TABLE PUTTING IN THE ENTRY FOR PCTARGET
         ETCRE ENTRIES=ETELIST        CREATE AN ENTRY TABLE
         ST R0,TKVALUE                GET THE ET TOKEN FROM REG 0
         LA R4,1
         ST R4,TKCOUNT                SET THE NUMBER OF ETS TO 1
*COMPUTE THE PC NUMBER (LX|EX) EX=0
          L R4,LXVALUE
          N R4,PCMASK                 CLEAR NON-LX VALUES
         ST R4,PC_NUMBER
         .
         .
*NEWADS CAN USE PC_NUMBER TO GET TO PCTARGET IN CREATOR FROM NEWADS
         ASCRE INIT='IEFBR14 ',ASNAME='NEWADS',TRMEXIT=TERMEXIT,       X
               AXLIST=AX_LIST,TKLIST=TOKENLIST,LXLIST=LX_LIST,         X
               ASPARM=PC_NUMBER,ODA=ODA_AREA
         .
*TERMINATION ROUTINE
TERMEXIT DS  0H
         USING  *,15               REGISTER 15 CONTAINS ENTRY ADDRESS
         SAVE   (14,12),,*         SAVE REGISTERS
         .
*PERFORM ADDRESS SPACE TERMINATION PROCESSING
         .
RETURN   (14,12)                   RESTORE REGISTERS; RETURN TO SYSTEM
         .
*DECLARATIONS
PCMASK    DC '000FFF00'             MASK TO CLEAR EX VALUE
PC_NUMBER DS F                      PC NUM USED TO GET TO PCTARGET
ODA_AREA  DS CL24                   OUTPUT DATA AREA
TOKENLIST DS CL8                    LIST OF ET TOKENS
LX_LIST   DS CL8                    LIST OF LXS
AX_LIST   DS CL4                    LIST OF AXS
    ORG LX_LIST
LXCOUNT   DS FL4                    NUMBER OF LX REQUESTED
LXVALUE   DS FL4                    ONE LX ENTRY
    ORG AX_LIST
AXENTRIES DS FL2                    NUMBER OF AX REQUESTED
AXENTRY   DS FL2                    ONE AX ENTRY
    ORG TOKENLIST
TKCOUNT   DS FL4                    NUMBER OF ET TOKENS
TKVALUE   DS FL4                    ET TOKEN
```

In the following example, a supervisor state key zero program deletes an address space that was created through the ASCRE macro. The address of the ASEO (ASE output area), passed back from ASCRE after creating the address space, is passed to this routine in general purpose register 1.

```
DESEXMPL CSECT
DESEXMPL AMODE 31
DESEXMPL RMODE ANY
         BAKR  14,0                  SAVE CALLER'S STATUS ON STACK
         BASR  12,0                  SET BASE REGISTER
         USING *,12
         USING ASEO,1                ESTABLISH ADDRESSABILITY TO ASEO
         .
         .
* DELETE THE ADDRESS SPACE
         ASDES STOKEN=ASEOSTKN
         .
* DECLARES THE DSECT FOR THE ASE OUTPUT AREA PASSED BACK FROM ASCRE.
* THE IHAASEO MACRO CONTAINS THIS MAPPING
         .
ASEO  DSECT
ASEOSTKN     DS XL8         64-BIT STOKEN OF THE NEW ASCB
ASEOASCB     DS A           ASCB OF NEW ADDRESS SPACE
```

```
ASEOECB      DS A           ECBs, BASING FOR IEZEAECB
ASEORSV1     DS XL8         RESERVED
```

Chapter 9. Creating and using subspaces

Within an application server address space, many application programs run under a single server program. An error in one of these application programs can cause it to overwrite the code or data of the other application programs or of the server program itself. **Subspaces** provide a means of limiting the application server address space storage that an application program can reference, thus limiting the damage an application program error can do within the application server address space.

This section describes the concept of subspaces, when to use them, how to create them, how to manage them, and how to delete them. It also describes considerations for providing recovery for and diagnosing errors in programs that run in subspaces.

What is a subspace?

A subspace is a specific range of storage in the private area of an address space, designed to limit the storage a program can reference.

A program that is associated with a subspace cannot reference some of the private area storage outside of the subspace storage range; the storage is protected from the program. Whether a given range of private area storage is protected from a program associated with a subspace depends on whether the storage is:

- Eligible to be assigned to a subspace (or **“subspace-eligible”**)
- Assigned to a subspace
- Not eligible to be assigned to a subspace.

You control these storage “states” through the IARSUBSP macro. Storage outside of the private area is not affected by subspaces.

A program running in an address space can reference all of the storage associated with that address space. In this section, a program's ability to reference all of the storage associated with an address space is called **full address space addressability.** A program running with full address space addressability can reference storage in any of the three states: eligible to be assigned to a subspace, assigned to a subspace, or not eligible to be assigned to a subspace.

A program that runs in an address space that owns subspaces also has full address space addressability. While running in a subspace, a program now has access to 64-bit private and shared storage. It can reference the 64-bit storage while in subspace mode and no longer needs to issue the BSG (Branch in Subspace Group) instruction to switch to the base mode to reference the 64-bit storage.

A program running in a subspace can reference storage that is assigned to its own subspace and storage that is not eligible to be assigned to a subspace. It cannot reference storage that is eligible to be assigned to a subspace or storage that is assigned to a subspace other than the one in which the program is running. In other words, a subspace allows a program running in it to reference all of the storage associated with the address space except the private area storage that is eligible to be assigned to a subspace or assigned to another subspace.

When storage is not eligible to be assigned to a subspace and not assigned to a subspace, it can be referenced by a program running in a subspace or a program running with full address space addressability. This storage **can be referenced by all subspaces** as well as by programs running with full address space addressability.

An address space that owns subspaces is also called a **“base space”**. Figure 68 on page 212 illustrates the concept of creating a subspace in base space ASID 23.

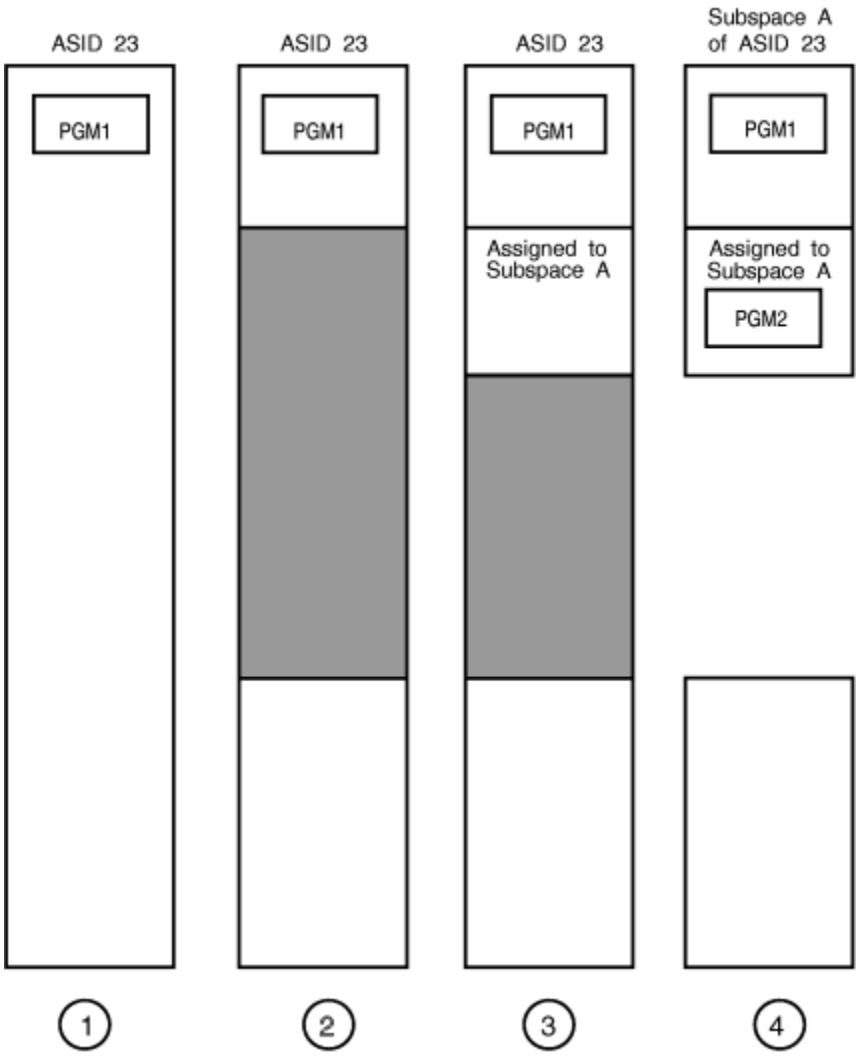


Figure 68. Illustration of address space that owns one subspace

1. PGM1 is a program running with full address space addressability in address space ASID 23. ASID 23 owns no subspaces, and no storage eligible to be assigned to a subspace. PGM1 can reference all storage in the address space.
2. PGM1 makes the shaded area of storage eligible to be assigned to a subspace. The eligible storage has not been assigned to a subspace. PGM1 can reference the subspace-eligible storage because PGM1 is not running in a subspace.
3. PGM1 assigns part of the subspace-eligible storage to Subspace A. PGM1 can reference the subspace storage as well as the subspace-eligible storage because PGM1 is not running in a subspace.
4. PGM1 issues the BSG instruction, which passes control to PGM2 to run in Subspace A. PGM2 can reference the storage that is assigned to Subspace A, and storage in the address space that has not been made subspace-eligible. PGM2 cannot reference the subspace-eligible storage while PGM2 is running in the subspace.

An address space can have many subspaces. Each application program running simultaneously in an address space can run in its own subspace. The subspace restricts a program running in it from referencing the storage assigned to other subspaces. Figure 69 on page 213 illustrates the concept of multiple subspaces by adding another subspace to address space ASID 23.

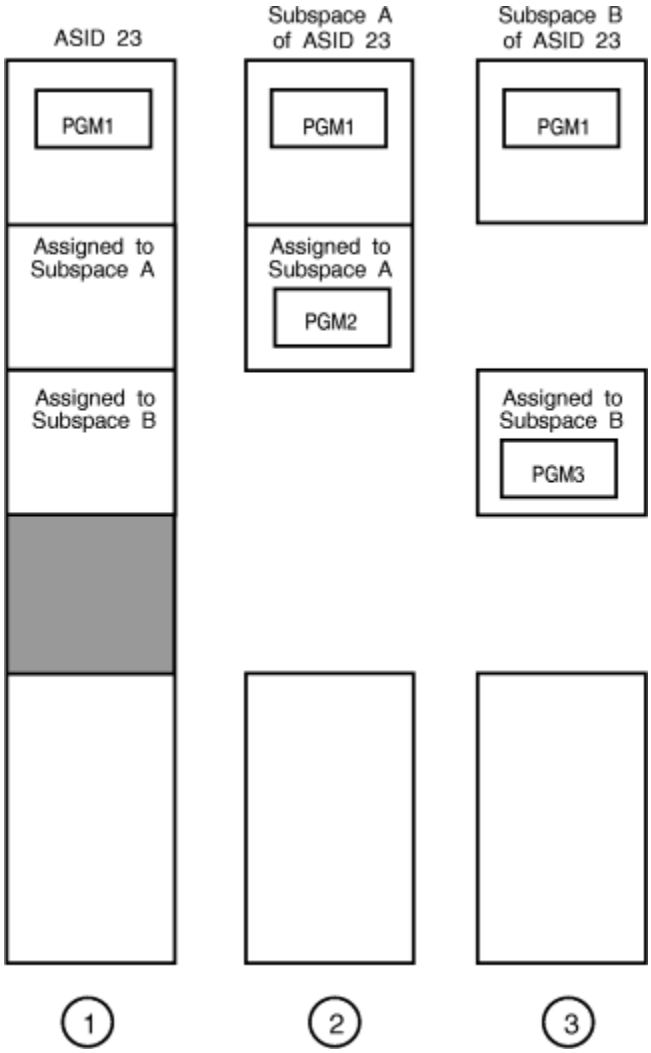


Figure 69. Illustration of address space that owns two subspaces

1. Running with full address space addressability, PGM1 creates and assigns storage to Subspace B. PGM1 can reference the entire address space, including storage assigned to Subspaces A and B, and subspace-eligible storage (shaded).
2. PGM2, running in Subspace A, can reference storage that is assigned to Subspace A and storage that has not been made subspace-eligible. PGM2 cannot reference storage in Subspace B or storage that is subspace-eligible.
3. PGM3 is a program running in Subspace B. PGM3 can reference storage that is assigned to Subspace B and storage in the address space that has not been made subspace-eligible. PGM3 cannot reference storage that is assigned to Subspace A, or storage that is subspace-eligible.

The number of subspaces per address space is limited by the amount of unallocated private storage available in the address space, and by the amount of storage assigned to each subspace.

A subspace is associated with only one address space and is owned by the task that creates it. A task cannot pass addressability to its subspaces to its subtasks or SRBs. An attached subtask or an SRB gets control with full address space addressability.

A subspace has an access list entry (called an "**entry**" in this section) associated with it. After a program creates a subspace, it adds the entry to the dispatchable unit access list (DU-AL) associated with the task the program runs under. A program does not have to be in AR mode to use a subspace, although it can be.

A program can toggle between running in a subspace and running with full address space addressability by issuing the BSG instruction.

Deciding whether your program should run in a subspace

Subspaces are beneficial in an application server address space in which numerous applications run under a single task within the address space.

Using subspaces as described in this section requires few or no changes to the application programs. See "Running a program in a subspace" on page 225 for some considerations for a program running in a subspace.

Using subspaces *does* require additional code in the server program. This requirement is explained in detail in "Steps to manage subspaces" on page 215.

Benefits of subspaces

The use of subspaces can protect the server and application programs in an address space. In addition, subspaces can help you to identify where in the address space an error has occurred.

Protecting the server program

Using subspaces in an application server address space protects the server program and its data. Subspaces reduce the number of failures in the server program by protecting it from the errors of other programs in the address space.

Protecting the application program

Using subspaces in an application server address space also protects the applications, similar to the way programs are protected by running in separate address spaces. By preventing applications from overwriting each other's code and data, subspaces increase the reliability of these applications.

Providing diagnosis

An IPCS diagnostic report and trace functions can help you to identify where in the address space an error has occurred.

An ABEND dump can help you to identify that an error resulted from a prohibited storage reference. When requested by a program running in a subspace, an ABEND dump contains only the storage that the program is allowed to reference.

Limitations of subspaces

Subspaces have the following limitations:

- They do not provide protection against deliberate attempts to overwrite code.
- To ensure that subspace storage is protected, the system abnormally ends a program that:
  - Attempts to reference storage to which it does not have addressability
  - Provides incorrect information on the IARSUBSP macro.

  Therefore, you might need to code additional recovery routines for your programs.
- An unauthorized application program running in a subspace cannot add more storage to the subspace. If an application program requires more subspace storage, the server program must obtain subspace-eligible storage for the application. This is explained in detail in "Requesting additional storage while running in a subspace" on page 226.
- Programs executing in an authorized state (for example, supervisor state, authorized key, or APF-authorized), might not always have subspace isolation and might have access to storage that is subspace-eligible but not assigned to the current subspace.

System storage requirements

One factor that might influence your decision to use subspaces is the amount of virtual and central storage that the system requires to manage them. This storage overhead can affect system performance.

The system allocates storage for its own use from subpool 255 when a program:

- Makes storage eligible to be assigned to a subspace
- Creates a subspace
- Assigns more than 2 segments of storage below 16 megabytes to a subspace.

The system deallocates its storage when the program deletes the subspace or makes the storage ineligible to be assigned to a subspace.

The amount of storage that the system requires for its own use depends on whether the subspace storage is above or below 16 megabytes. The system requires more storage to manage subspaces below 16 megabytes.

Use the following guidelines to plan for the system's storage requirements:

Table 25. System storage requirements when managing subspaces

The System Uses:	**For Each:**
8192 bytes	Address space in which a program issues IARSUBSP IDENTIFY
1024 bytes	Segment below 16 megabytes specified on IARSUBSP IDENTIFY
10376 bytes	Subspace created
1024 bytes	Segment below 16 megabytes specified on IARSUBSP ASSIGN, **after the first two such segments**

Steps to manage subspaces

The steps to create, branch to, and delete subspaces, and the macros and instructions associated with each step, are outlined in Table 26 on page 215. The table also includes the program authorization requirements. Each step is explained in detail on the topic indicated.

Table 26. Steps for creating, using, and deleting subspaces

Step	**Minimum Authorization**	**Performed by:**	**See**
Determine if the subspace is available on your system	Problem state and any PSW key	Testing a bit in the CVT	"Determining whether subspaces are available on your system" on page 218
Obtain storage for subspaces		Using the GETMAIN or STORAGE macro	"Obtaining storage for subspaces" on page 219

Table 26. Steps for creating, using, and deleting subspaces (continued)

Step	Minimum Authorization	Performed by:	See
Make the storage eligible to be assigned to subspaces	Supervisor state or PSW key 0 - 7	Using the IARSUBSP macro with IDENTIFY	"Making a range of storage eligible to be assigned to a subspace" on page 220
Create the subspaces		Using the IARSUBSP macro with CREATE	"Creating the subspaces" on page 222
Add the subspace entries to the DU-AL	Problem state and any PSW key	Using the ALESERV macro with ADD	"Establishing addressability to a subspace" on page 223
Assign the identified storage to the subspace		Using the IARSUBSP macro with ASSIGN	"Assigning storage to the subspaces" on page 223
Branch and run an application program in a subspace		Using the BSG instruction	"Branching to a subspace" on page 224
Disassociate the storage from the subspace		Using the IARSUBSP macro with UNASSIGN	"Disassociating storage from the subspaces" on page 227
Remove the entry from the DU-AL		Using the ALESERV macro with DELETE	"Removing the subspace entry from the DU-AL" on page 227
Delete the subspace	Supervisor state or PSW key 0 - 7	Using the IARSUBSP macro with DELETE	"Deleting the subspace" on page 228
Make the storage ineligible to be assigned to a subspace.		Using the IARSUBSP macro with UNIDENTIFY	"Making storage ineligible to be assigned to a subspace" on page 228
Release the storage	Problem state and any PSW key	Using the FREEMAIN or STORAGE macros	"Releasing storage" on page 228

Updating the application server to use subspaces

Most application servers consist of at least two types of programs:

- **Application programs**, which perform the work
- **Server programs**, which manage the application programs and the address space.

You can choose to manage subspaces in either of the following ways, or with a combination of the two:

- Create a number of subspaces prior to receiving requests for application program services
- Create one subspace at a time, in response to receiving a request for application program services.

The method that you choose depends on whether your installation is more concerned with storage constraints or performance of the application server.

Managing subspaces when performance is a priority

It is most efficient to obtain storage for and create the number of subspaces needed for all application programs as part of application server initialization. Then, as a request for an application program's services is received, the server program assigns eligible storage to a subspace, runs the application program in the subspace, and disassociates the eligible storage from the subspace. As part of application server termination, the application server deletes the subspaces and makes the storage ineligible to be assigned to a subspace.

Managing subspaces in this way is less costly than other designs in terms of performance. The IDENTIFY and CREATE functions use more instructions than the ASSIGN and UNASSIGN functions. By reserving a quantity of subspace-eligible storage and creating subspaces that are reused for multiple invocations of the application programs, the server program manages subspaces efficiently.

This design could cause storage constraints. When storage is subspace-eligible but not assigned to a subspace, a program running in a subspace cannot reference it. Subspace-eligible storage cannot be released until the server program makes it ineligible to be assigned to a subspace. Furthermore, a program cannot pass ownership of subspace-eligible storage to a subtask.

If storage constraints in the application server address space are a concern at your installation, you might want to consider the alternate design described in “Managing subspaces when storage is a priority” on page 217.

Managing subspaces when storage is a priority

A server program with storage constraints can manage the subspaces by performing all steps to create and delete subspaces each time an application program runs. (See Table 26 on page 215 for an overview of the steps.) Managing subspaces in this way can reduce storage contention in the system, but is much more costly in terms of server performance.

Creating a single subspace

The following is a simple illustration of how a server program can manage a single subspace.

Table 27. How a server program manages single subspaces.

Server program	Functions
STORAGE OBTAIN	Obtain storage in application server address space Receive storage to be used for subspaces
IARSUBSP IDENTIFY	Make storage ranges eligible for subspaces Specify storage that was previously obtained

Table 27. How a server program manages single subspaces. (continued)

Server program	Functions
IARSUBSP CREATE	Create the subspace Receive STOKEN
ALESERV ADD	Add the subspace to the DU-AL, specifying STOKEN Receive ALET
IARSUBSP ASSIGN	Assign the range of storage that a program running in the subspace can reference Specify STOKEN, storage portion
BSG	Branch to subspace Specify ALET
	Run application program in subspace
BSG	Branch back to full address space addressability Specify ALET 0
IARSUBSP UNASSIGN	Disassociate the range of storage from the subspace Specify STOKEN, storage range
ALESERV DELETE	Remove entry from the access list Specify ALET
IARSUBSP DELETE	Delete the subspace Specify STOKEN
IARSUBSP UNIDENTIFY	Make storage ranges ineligible for subspace usage Specify storage that was previously specified on IARSUBSP IDENTIFY
STORAGE RELEASE	Release storage in application server address space Specify storage

Determining whether subspaces are available on your system

Before attempting to use subspaces, your program should ensure that the subspace is installed on your system. To test for the subspace, include the CVT in your program and check the CVTSUBSP bit. When this bit is on, the subspace is available on your system.

Obtaining storage for subspaces

After determining that the subspace is available, the server program must obtain storage for the subspaces. As explained in "Steps to manage subspaces" on page 215, it is most efficient to obtain in one request enough storage for all subspaces that the application programs will require. You will need the following information about the application programs running in the address space to estimate the size of your storage request:

- The average number of application programs in the application server address space during peak processing periods
- The average amount of storage required for an application program and its data
- The amount of "surplus" storage you want available to be used by application programs during unusually heavy workloads or for large application programs.

This information might be available from a performance monitoring program. If not, you might want to estimate the storage required and fine-tune the storage request later, after testing your estimate.

In addition, be sure that you request enough storage to allow you to align the storage on a megabyte boundary. To align the storage correctly, you might have to request a good deal more storage than you plan to use.

Storage attributes

Obtain your subspace storage by selecting a storage subpool with the storage attributes that subspaces require. The section on virtual storage in *z/OS MVS Programming: Authorized Assembler Services Guide* contains a table listing all subpools and the storage attributes associated with them. Table 28 on page 219 shows the required storage attributes for subspaces.

Table 28. Storage attributes required for subspaces

Storage Attribute	Requirement	Comments
Location	Private	Subspace storage must be in high private or low private storage.
Fetch Protection	None	Subspace storage can be fetch-protected, but fetch-protection is not required.
Type	Pageable	Subspace storage must be pageable.
Owner	Task or job step	Subspace storage must be owned by the task creating the subspace, or a task higher in the task hierarchy.
Storage key	None	Subspace storage has no storage key requirements.

Backing virtual storage for a subspace

Subspaces can be backed by real storage either above or below 16 megabytes. Backing the subspace below 16 megabytes is more costly in the event of a page fault. Back a subspace with storage above 16 megabytes unless an old application, which must run below 16 megabytes, will run in the subspace.

Requesting subspace storage

Use the STORAGE macro to request storage for your subspaces. You can also use the GETMAIN macro, but STORAGE is easier to use and has fewer restrictions and requirements than GETMAIN.

When the STORAGE macro successfully obtains storage, it returns the length and address of the storage. You supply the length and address of the obtained storage in a **range list** when you invoke the IARSUBSP macro to make the storage eligible to be assigned to a subspace. (Making storage eligible is described in detail in "Making a range of storage eligible to be assigned to a subspace" on page 220.)

Aligning virtual storage for a subspace

After you obtain the storage for the subspaces, you must align the storage on a megabyte boundary before specifying it on an IDENTIFY request. See “Example of managing subspaces” on page 228, which illustrates how to align the storage you've obtained.

Creating the range list

The range list is a storage area containing up to 16 entries. Each entry consists of 2 words as follows:

First word
The starting virtual address of the storage range that the system is to make eligible to be assigned to a subspace. The starting address must be on a megabyte boundary. A megabyte is 1,048,576 bytes long.

Second word
The number of pages (4096 bytes), beginning at the address in the first word, that are to be made eligible to be assigned to a subspace. The number must be a multiple of 256.

The range list must be addressable in the caller's primary address space. Each range must reside in a single subpool.

You might not be able to obtain all of the subspace storage required by your application programs in a single STORAGE macro request. If you cannot, add an entry to the range list for each storage request. The STORAGE macro returns the number of bytes of storage obtained in GPR 0. The second word of the range list entry requires the number of pages obtained. To convert the number of bytes into the number pages, divide the number of bytes returned in GPR 0 by 4096. Store the quotient into the second word of the range list entry. Store the contents of GPR 1 into the first word of the range list entry.

Making a range of storage eligible to be assigned to a subspace

After the server program has obtained storage, it must make the storage eligible to be assigned to a subspace. To do this, invoke the IARSUBSP macro with the IDENTIFY parameter, specifying the storage range.

A program that is running in a subspace cannot reference a range of storage once the storage range is eligible to be assigned to a subspace. If it attempts to do so, it will abnormally end with system completion code X'0C4'. In addition, the server program cannot pass ownership of this subspace-eligible storage to a subtask. If the server program attempts to do so by invoking the ATTACH macro with either the GSPL or GSPV parameter, the system will abnormally end the server program with system completion code X'12A'.

A server program that attempts to release the storage before the storage has been made ineligible to be assigned to a subspace will abnormally end with system completion code X'A05', X'A0A', or X'A78'. To make the storage ineligible to be assigned to a subspace, specify the storage range on the IARSUBSP macro with the UNIDENTIFY parameter.

Considerations when making storage eligible to be assigned to a subspace

When updating the server to make storage eligible to be assigned to a subspace, consider the task under which the server is running, and the restrictions of programs running in subspaces.

Task hierarchy restrictions

Your server program should run under the lowest task in the task hierarchy that will need to make storage eligible to be assigned to a subspace.

The first time a server program successfully issues the IARSUBSP IDENTIFY request in an address space, the system identifies the task under which that program runs as the lowest task in the task hierarchy that can make subsequent IARSUBSP IDENTIFY requests. Additionally, that task or a higher task must own the storage that is being made eligible to be assigned to a subspace.

Effect on existing subspaces

A program running in a subspace cannot reference storage once it has been made eligible to be assigned to a subspace. If an address space already owns subspaces and makes additional storage eligible to be assigned to a subspace, the programs running in the existing subspaces lose the ability to reference the storage that has been made subspace-eligible. The effect is that a program running in a subspace becomes unable to reference storage that it could reference prior to the IDENTIFY request.

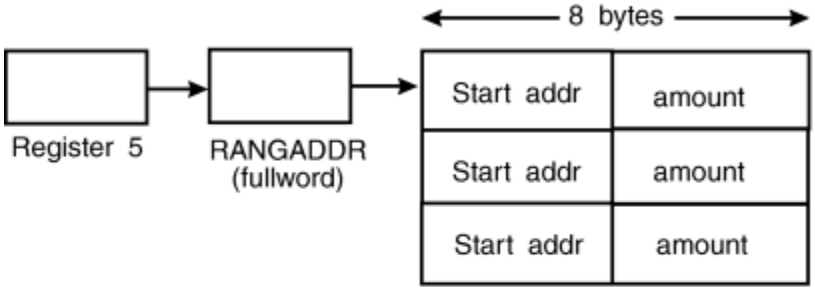
Coding the RANGLIST parameter

Together with the NUMRANGE parameter, the RANGLIST parameter allows you to make multiple storage ranges eligible to be assigned to subspaces. The RANGLIST parameter specifies a fullword that contains an **address of** a list of ranges, or specifies a register that contains the address of the fullword pointer to the range list that you created when you allocated the storage. The number of entries in the range list is specified on the NUMRANGE parameter.

The following examples illustrate the range list and how to use a register or a fullword field as pointers to it.

The range list contains 3 entries and RANGLIST uses register notation:

NUMRANGE=3,RANGLIST=(5)



The range list contains 3 entries and RANGLIST uses a fullword pointer:

NUMRANGE=3,RANGLIST=RANGADDR

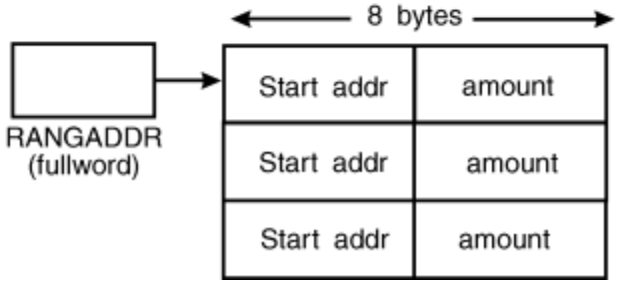


Figure 70. Illustration of the range list

Requirements of the range list for an IDENTIFY or UNIDENTIFY request

Each range list entry representing storage for an IDENTIFY or UNIDENTIFY request must:

- Specify a number of pages that is a multiple of 256
- Specify an address that begins on a segment boundary
- Specify storage that previously has been obtained

- Specify storage that is pageable and private
- Specify storage that is owned by the task that previously invoked IARSUBSP IDENTIFY, or by a task that is higher in the task hierarchy.

For the requirements of the range list for an ASSIGN or UNASSIGN request, see “Requirements of the range list for an ASSIGN or UNASSIGN request” on page 224.

System processing of range list errors in IARSUBSP IDENTIFY request

If an entry in the range list does not conform to one or more of these requirements, the system processes the range list entries up to the entry in error. The system does not process the incorrect range list entry or any range list entries that follow it. The system abnormally ends the IARSUBSP IDENTIFY request with system completion code X'3C6', and puts the address of the incorrect range list entry in GPR 2. It puts the address of the storage that incurred the error into GPR 3.

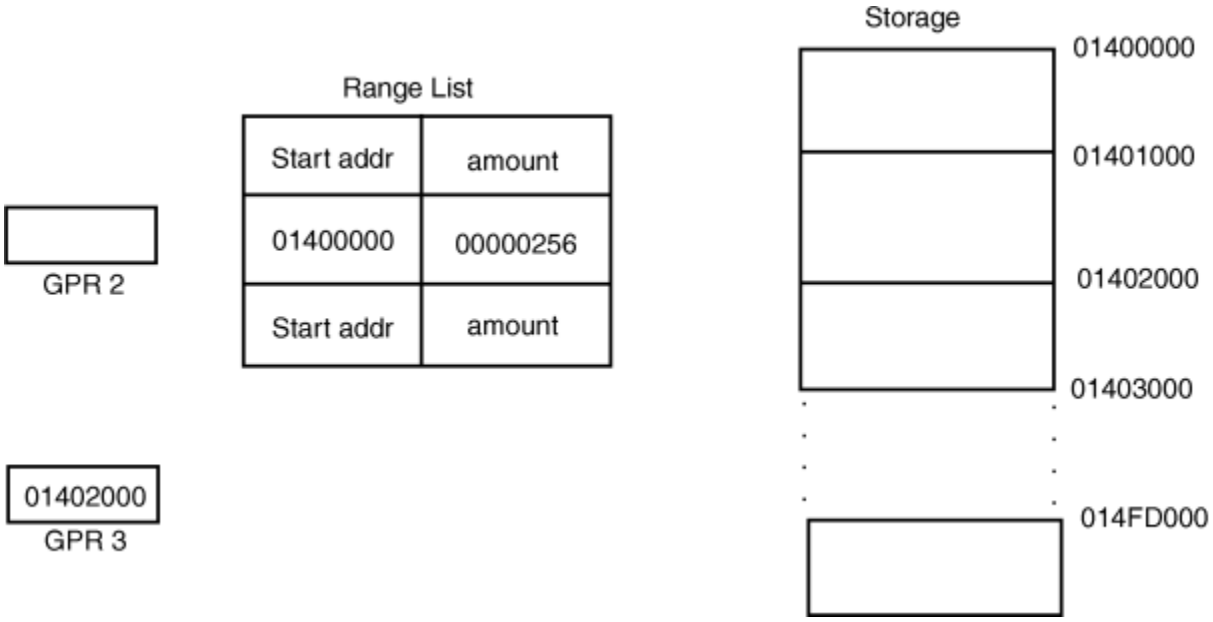


Figure 71. Illustration of GPR Contents in Event of Range List Error

Creating the subspaces

The server program can create subspaces by issuing the IARSUBSP macro with the CREATE parameter. IARSUBSP CREATE allows you to name the subspaces yourself, have the system generate the subspace names, or have the system generate subspace names only when the name you provide is not unique within the address space. See the IARSUBSP macro description in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for more information about naming subspaces.

Be aware that, if you choose to let the system generate the subspace names for you, you must still supply three characters for the system to use.

Saving subspace STOKENs

When it creates a subspace, the system returns an STOKEN for the subspace. Save the subspace STOKEN returned to your program. You must provide the STOKEN when adding an access list entry for the subspace. A program cannot run in a subspace until the server program uses the STOKEN to add an access list entry to its task's DU-AL.

You also must supply the subspace STOKEN to associate and disassociate storage with a subspace, and to delete the subspace after your application program has run.

Establishing addressability to a subspace

Before a program can run in a subspace, the server program must add a subspace entry to the DU-AL of the task that the program is running under. To do this, code the ALESERV macro with the ADD parameter, supplying the STOKEN that the system returned from the IARSUBSP CREATE request.

The requirements for the ALESERV request are:

- The subspace entry must be added as a public entry. If you attempt to add it as a private entry, you will receive a return code indicating an error from the ALESERV macro.
- The subspace entry must be added to the DU-AL. If you attempt to add a subspace entry to the PASN-AL, you'll receive a return code indicating an error from the ALESERV macro.

The ALESERV macro returns an ALET. Do not modify this ALET. Use the ALET as an operand on the BSG instruction to switch to a subspace.

Copying the DU-AL to a subtask or SRB

Although a program can request that the system provide a subtask or SRB with a copy of its DU-AL, the system cannot copy subspace entries to the new DU-AL. The copy of the DU-AL will contain all of the access list entries *except* those representing subspaces.

Searching for and extracting a subspace entry

Once a subspace entry has been added to a DU-AL, a program can use the SEARCH parameter of ALESERV to obtain the subspace ALET, and the EXTRACT parameter of ALESERV to obtain the subspace STOKEN.

A program cannot invoke ALESERV EXTRACT and supply ALET 1.

Using special alets

The meaning of special ALETs 0 and 1 differs depending on whether the ALET is supplied for access register translation or specified on the BSG instruction.

A program running in AR ASC mode in a subspace or with full address space addressability can use special ALETs 0, 1, and 2 to access its primary, secondary, and home address spaces. This concept is described in detail in “Special ALET values” on page 103.

Likewise, a program running in a subspace or with full address space addressability can supply these ALETs on an ALESERV EXTRACT request to obtain the STOKEN for the address space they require. To obtain the STOKEN for the full address space, invoke ALESERV with the EXTRACTH parameter.

However, when specified on a BSG instruction, ALETs 0 and 1 have different meanings. Specifying BSG with ALET 0 causes a program to run with full address space addressability. Specifying BSG with ALET 1 causes a program to run in the subspace that most recently had control. If a program is running in a subspace when it issues BSG with ALET 1, the program will continue to run in that subspace.

If a program has never run in a subspace and it attempts to return to a subspace by issuing the BSG instruction with ALET 1, the program will abnormally end with system completion code X'0D3'.

Assigning storage to the subspaces

Before a program can reference subspace storage, the program must associate an eligible range of storage with the subspace. Storage is eligible to be assigned to a subspace once it has been specified on an IARSUBSP IDENTIFY request. Storage that has been assigned to a subspace can be referenced only by a program running in that subspace, or by a program running with full address space addressability.

Use the IARSUBSP macro with the ASSIGN parameter to associate a subspace with its storage. Either an authorized or unauthorized program can perform this step. Specify a storage range from the storage you obtained, and the STOKEN returned when you created the subspace.

One way to design your server program is to create a loop in the server program. For each request to the server program for application program services, the program loop:

1. Assigns a storage range to a subspace
2. Issues the BSG instruction to switch to subspace addressability
3. Passes control to the application program
4. Receives control after the application program finishes its processing
5. Issues the BSG instruction to switch to full address space addressability
6. Disassociates the storage from the subspace (described in more detail in "Disassociating storage from the subspaces" on page 227).

This design ensures that there is a subspace available for the application program to run in. It also allows eligible storage to be reassigned to different subspaces as needed, while preventing application programs from referencing the storage as it is being reassigned.

Requirements of the range list for an ASSIGN or UNASSIGN request

The requirements for a range list entry for an ASSIGN or UNASSIGN request differ depending on whether the storage the entry represents is above or below 16 megabytes.

For storage above 16 megabytes, a range list entry must:

- Specify a number of pages that is a multiple of 256
- Specify an address that begins on a segment boundary
- Specify storage that previously has been obtained
- Specify storage that has been made eligible to be assigned to a subspace by a previous IARSUBSP IDENTIFY request.

A range list entry representing storage below 16 megabytes for an ASSIGN or UNASSIGN request must:

- Specify a number of pages
- Specify an address that begins on a page boundary
- Specify storage that has previously been obtained
- Specify storage that has been made eligible to be assigned to a subspace by a previous IARSUBSP IDENTIFY request.

System processing of range list errors in IARSUBSP ASSIGN request

If the storage range you specify is already assigned to a subspace, the system does not process that request but will continue to process the subsequent valid range list entries. The system places return code 4 in GPR 0.

If an entry in the range list does not conform to one or more of the range list requirements, the system processes the range list entries up to the entry in error. The system does not process the incorrect range list entry or any range list entries that follow it. The system abnormally ends the IARSUBSP ASSIGN request with system completion code X'3C6', and puts the address of the incorrect range list entry in GPR 2. It puts the address of the storage range into GPR 3.

Branching to a subspace

A server program branches to a subspace when it issues the BSG instruction with the ALET that corresponds to the desired subspace, and its primary and home address spaces are the same. The BSG instruction uses an ALET-qualified branch address to switch to the subspace. Use the ALET returned by ALESERV ADD.

The server program can use the BSG instruction to pass control to an application program which will run in the subspace. Alternately, the server program itself can begin running in the subspace. See *Principles of Operation* for details about coding the BSG instruction.

After issuing the BSG instruction, a program can reference the subspace while running in primary, secondary, or AR modes. The program runs in a subspace until it switches to:

- Another subspace or full address space addressability, by issuing another BSG instruction
- Another address space, by issuing a space-switching instruction.

Using cross memory mode with subspaces

If the program changes its primary or secondary address space to be other than the home address space, it loses the ability to address the subspace. It can regain addressability to the subspace by setting the changed address space back to the home address space.

Example of changing primary and secondary address spaces with subspaces

An application program is running with PASN=SASN=HASN in address space X'23', and has not issued the BSG instruction.

Here are some hypothetical actions and the results of those actions:

Program Action	Result
The program issues a BSG to subspace TP1	The program is running with PASN=SASN=HASN=X'23', and has addressability to subspace TP1 through its primary and secondary address spaces.
The program issues a PC (Program Call) instruction that changes the PASN to X'14'	The program is running with PASN=X'14', SASN=HASN=X'23', and has addressability to subspace TP1 through its secondary address space.
The program issues a SSAR (Set Secondary ASN) instruction that changes SASN to ASID X'18'	The program is running with PASN=X'14', SASN=X'18', HASN=X'23', and does not have addressability to TP1 through either its primary or secondary address spaces.
The program issues a SSAR instruction that changes SASN back to ASID X'23'	The program is running with PASN=X'14', SASN=HASN=X'23', and has addressability to subspace TP1 through its secondary address space.
The program issues a PT (Program Transfer) instruction that changes PASN back to ASID X'23'	The program is now running with PASN=SASN=HASN=X'23' and has addressability to subspace TP1 through its primary and secondary address spaces.
The program issues a BSG instruction using ALET 0	The program is running with PASN=SASN=HASN=X'23' and has full address space addressability.

Running a program in a subspace

A program running in a subspace will abnormally end if it attempts to reference:

- Storage that is assigned to another subspace
- Storage that is eligible to be assigned to another subspace, but is not assigned.

Aside from these restrictions, a program running in a subspace can reference the same storage that a program running with full address space addressability can reference.

The following topics describe additional considerations for a program running in a subspace.

Authorized programs and subspaces

Subspaces are intended to give storage isolation for unauthorized programs; however, programs that are authorized (for example, supervisor state, authorized key, or APF-authorized) might not always have the same isolation. When running as authorized in a subspace, a program might have access to storage that is subspace-eligible but not assigned to the current subspace. As with any authorized program, those running in a subspace should ensure that storage boundaries are properly validated when parameters are passed from an unauthorized source.

Returning to full address space addressability

At any time during processing, a program can return to full address space addressability. If the primary address space is other than the home address space, the program must first change its primary address space to its home address space. Then, the program can issue the BSG instruction using ALET 0 to return to full address space addressability. Running with full address space addressability allows the program to reference the address space private storage without regard to subspaces.

Preserving the path across subspaces

The BSG instruction allows you to return to the subspace in which the program last ran by issuing BSG with ALET 1. However, BSG cannot reconstruct a program's path across multiple subspaces. It can return the program only to the last subspace it was in. The program is responsible for preserving the subspace trail, if it needs that information.

Using checkpoint/restart with subspaces

A program cannot request a checkpoint while running in a subspace. A program running with full address space addressability can request a checkpoint if it has no subspace entries on its DU-AL, or if it ensures that the system will ignore the subspace entries by either:

- Deleting all DU-AL entries that were not created specifying CHKPT=IGNORE on the ALESERV request, including subspace entries.
- Ensuring that the access list entries were added by specifying CHKPT=IGNORE on the ALESERV request.

A task that either deletes the entries or adds them specifying CHKPT=IGNORE is responsible for rebuilding the subspaces and reestablishing the connections to them. If a restart occurs after a successful checkpoint, the system does not rebuild the subspaces or establish addressability to them.

Requesting additional storage while running in a subspace

If a program running in a subspace needs additional storage, you must determine whether that storage must be protected by the subspace. If the storage does not need subspace protection, the program can obtain it by using the STORAGE macro and requesting storage that is not eligible to be assigned to a subspace.

A program running in the subspace might use storage that is not protected by a subspace to share data with a program running in another subspace, or to provide access to parameter lists, data areas, or exits needed by an MVS service.

If the additional storage must be protected by the subspace, the application program must have the server program obtain storage on its behalf. The server program can use the surplus storage that it obtained (described in “Obtaining storage for subspaces” on page 219). If none is available, the server program:

1. Makes a request for more storage meeting the requirements described in “Storage attributes” on page 219
2. Makes the storage range eligible to be assigned to a subspace by specifying the range on IARSUBSP IDENTIFY

3. Assigns the storage to the subspace by specifying the storage range and the subspace STOKEN on the IARSUBSP ASSIGN request.

After doing so, the server program can pass control back to the application program, which can then use the additional storage.

Keep in mind that the server program should preserve the starting address and number of pages of any additional storage that it obtains, to disassociate the storage and make it ineligible to be assigned to a subspace when the application program has finished processing, and to release the storage.

Using MVS services in a subspace

A program that uses MVS services while running in a subspace must have storage access to the service. For example, if a program loads a copy of an MVS service, it must ensure that the load module is loaded into either:

- Storage that is assigned to the subspace
- Storage that can be referenced by all subspaces (storage that has not been specified on an IARSUBSP IDENTIFY request).

Additionally, the program must ensure that the MVS service has access to all required parameter lists, data areas, and program exits, by keeping them in storage that is assigned to the subspace or storage that can be referenced by all subspaces. If the program cannot provide access to both of these storage areas, it might have to switch to full address space addressability to use the MVS service.

Finally, the program must ensure that all necessary storage is available to an MVS service across asynchronous operations.

Disassociating storage from the subspaces

After the application program has run, the server program can disassociate the subspace-eligible storage from the subspace to which it is assigned. This allows the server program to assign eligible storage to another subspace when it receives a new request for application program services. Disassociating the storage also prevents an application program from referencing the storage before it is reassigned.

Use the IARSUBSP macro with the UNASSIGN parameter to disassociate subspace-eligible storage from the subspace to which it is assigned.

System processing of range list errors in IARSUBSP UNASSIGN request

The storage ranges supplied in the range list for an IARSUBSP UNASSIGN request must meet the requirements described in “Requirements of the range list for an ASSIGN or UNASSIGN request” on page 224.

If the storage range you specify is not assigned to the subspace you specify, the system does not process that request but will continue to process the subsequent valid range list entries. The system places return code 4 in GPR 0.

If an entry in the range list does not conform to one or more of the range list requirements, the system processes the range list entries up to the entry in error. The system does not process the incorrect range list entry or any range list entries that follow it. The system abnormally ends the IARSUBSP UNASSIGN request with system completion code X'3C6', and puts the address of the incorrect range list entry in GPR 2. It puts the address of the storage range into GPR 3. See Figure 71 on page 222 for an illustration of a range list error.

Removing the subspace entry from the DU-AL

Prior to deleting the subspace, the server program should remove the subspace's associated entry from the DU-AL. Do this by invoking the ALESERV macro with the DELETE parameter and the subspace STOKEN.

If the task that created the subspace ends before the subspace entry has been deleted, the system will remove the entry from the DU-AL.

Deleting the subspace

An authorized program can delete a subspace by invoking the IARSUBSP macro with the DELETE parameter and supplying the subspace STOKEN that the system returned when the program created the subspace. It is most efficient to delete all subspaces at once.

The program that deletes a subspace must be running under the same task as the program that created the subspace. The program will abnormally end if it attempts to delete a subspace that it, or any other program, is running in.

The system disassociates the storage from the subspace before deleting the subspace, if the program has not already done so.

Deleting a subspace does not remove its associated entry from the DU-AL. See "Removing the subspace entry from the DU-AL" on page 227 for information about deleting the entry.

Making storage ineligible to be assigned to a subspace

All storage that has been made eligible to be assigned to a subspace must be specified on an IARSUBSP UNIDENTIFY request before it can be released. The server program must invoke the IARSUBSP macro with the UNIDENTIFY parameter, and specify the storage range in the range list.

The system disassociates the storage from the subspace before making the storage ineligible to be assigned to a subspace, if the server program has not already done so.

System processing of range list errors in IARSUBSP UNIDENTIFY request

The storage ranges supplied in the range list for an IARSUBSP UNIDENTIFY request must meet the requirements described in "Requirements of the range list for an IDENTIFY or UNIDENTIFY request" on page 221. In addition, the storage range must be eligible to be assigned to a subspace.

If an entry in the range list does not conform to one or more of these requirements, the system processes the range list entries up to the entry in error. The system does not process the incorrect range list entry or any range list entries that follow it. The system abnormally ends the IARSUBSP UNIDENTIFY request, with system completion code X'3C6', and puts the address of the incorrect range list entry in GPR 2. It puts the address of the storage range into GPR 3.

Releasing storage

A program cannot release the storage that was obtained for subspaces until it issues the IARSUBSP macro with the UNIDENTIFY parameter. If it attempts to free the storage before issuing IARSUBSP UNIDENTIFY, the program will abnormally end with system completion code A05, A0A, or A78. See *z/OS MVS System Codes* for information about those codes.

Use the STORAGE macro with the RELEASE parameter to release the storage that you obtained for the subspaces. You can also use the FREEMAIN macro, but STORAGE has fewer requirements and restrictions and is easier to use.

Free the storage by specifying on the SP parameter the subpools you obtained for your subspace storage.

Example of managing subspaces

```
*     OBTAIN THE STORAGE FROM A PAGEABLE SUBPOOL
         STORAGE OBTAIN,LENGTH=4096*(256+256),BNDRY=PAGE,SP=0,COND=YES
         ST    1,STORSTRT
         LTR   15,15                     IF NOT SUCCESSFUL (0)
         BNZ   NOSTOR                    GO TO ERROR PROCESSING
*
*     MAKE IT SEGMENT ALIGNED
*
         L     9,ROUNDIT
```

```
         L     2,ONEMEG
         L     10,STORSTRT
         ALR   10,2
         NR    10,9
         ST    10,STORSEGA             NEW SEGMENT-ALIGNED BOUNDARY
         L     1,STORSEGA
         ST    1,RPTR1                 PUT IT IN THE RANGE LIST
*  ******************************************************************
*     CREATE 5 SUBSPACES
*  ******************************************************************
         LA    5,1                     INIT LOOP COUNTER
         LA    9,STOKEN1               START WITH FIRST STOKEN
*                                     IN ARRAY
LOOP1    DS    0H
         IARSUBSP CREATE,NAME=SSNAME,STOKEN=(9),                       *
               GENNAME=COND,OUTNAME=ONAME
         LTR   15,15                   IF NOT SUCCESSFUL (0)
         BNZ   NOCREATE                GO TO ERROR PROCESSING
         LA    4,1                     LOOP INCREMENT IS 1
         ALR   5,4                     BUMP UP LOOP COUNTER
         LA    10,8                    ARRAY INCREMENT IS 8
         ALR   9,10                    BUMP UP ARRAY INDEX
         LA    4,5
         CR    5,4                     CHECK HOW MANY SO FAR
         BNH   LOOP1                   IF NOT 5 YET, REPEAT
 *  ******************************************************************
*     ADD THE SUBSPACE ENTRY TO THE WORKUNIT ACCESS LIST
*  ******************************************************************
         ALESERV ADD,STOKEN=STOKEN1,ALET=SSALET,AL=WORKUNIT
*  ******************************************************************
*     MAKE THE STORAGE SUBSPACE-ELIGIBLE
*  ******************************************************************
         IARSUBSP IDENTIFY,RANGLIST=RANGPTR,NUMRANGE=NUMRANG
*  ******************************************************************
*     ASSIGN THE STORAGE TO THE SUBSPACE
*  ******************************************************************
         IARSUBSP ASSIGN,STOKEN=STOKEN1,RANGLIST=RANGPTR
*  ******************************************************************
*     BRANCH TO THE SUBSPACE
*  ******************************************************************
          L     2,=A(X'80000000'+NEXT1)
          BSG   0,2
```

```
*  ******************************************************************
*     RUN PROGRAM IN THE SUBSPACE
*  ******************************************************************
*    RETURN TO THE BASE SPACE (FULL ADDRESS SPACE ADDRESSABILITY)
*  ******************************************************************
NEXT1     DS    0H
          L     0,=A(X'80000000'+NEXT2)
          BSG   0,0
*  ******************************************************************
*  DISASSOCIATE THE STORAGE (NUMRANGE DEFAULTS TO 1 WHICH IS WHAT
*  WE HAVE)
*  ******************************************************************
NEXT2     DS    0H
         IARSUBSP UNASSIGN,STOKEN=STOKEN1,RANGLIST=RANGPTR
*  ******************************************************************
*  MAKE THE STORAGE INELIGIBLE TO BE ASSIGNED TO A SUBSPACE
*  ******************************************************************
         IARSUBSP UNIDENTIFY,RANGLIST=RANGPTR
*  ******************************************************************
*  DELETE THE SUBSPACE
*  ******************************************************************
*
         IARSUBSP DELETE,STOKEN=STOKEN1
*
*  ******************************************************************
*  SUBSPACE CREATE FAILED - RELEASE THE STORAGE
*  ******************************************************************
NOCREATE DS    0H                     ERROR EXIT POINT
*  ******************************************************************
*  RELEASE THE STORAGE - USE THE ORIGINAL ADDRESS STORSTRT
*  ******************************************************************
         STORAGE RELEASE,ADDR=STORSTRT,LENGTH=4096*(256+256)
*  ******************************************************************
*  STORAGE OBTAIN FAILED - UNDO WHATEVER STEPS HAD BEEN SUCCESSFUL
*  PRIOR TO THE STORAGE OBTAIN
*  ******************************************************************
NOSTOR   DS    0H                     Error exit point
          .
```

```
          .
          .
          .
          .
          .
```

```
*  ********************************************************************
*  DECLARES
*  ********************************************************************
ONEMEG   DC    F'1048576'              ONE MEGABYTE
ROUNDIT  DC    X'FFF00000'             ROUND IT TO A SEGMENT ADDRESS
SSNAME   DC    CL8'SSPACE1 '           SUBSPACE NAME
ONAME    DS    CL8                     GENERATED NAME IF NEEDED
SSSTOKEN DS    0CL40
STOKEN1  DS    CL8
STOKEN2  DS    CL8
STOKEN3  DS    CL8
STOKEN4  DS    CL8
STOKEN5  DS    CL8
SSALET   DS    5CL4
STORSTRT DS    1F                      ADDRESS FOR OBTAIN/RELEASE
STORSEGA DS    1F                      SEGMENT-ALIGNED ADDRESS
*
* RANGE LIST MAPPING
*
RLIST    DS    0CL8
RPTR1    DS    F
NUMBLKS  DC    F'256'
*
RANGPTR  DC    A(RLIST)
NUMRANG  DC    F'1'
```

Planning for recovery in a subspace environment

As described in “Limitations of subspaces” on page 214, the system abnormally ends programs that specify incorrect parameters on the IARSUBSP macro. While this helps to preserve the integrity of subspaces, the chances that your server program will abnormally end are increased.

You can plan for this by designing recovery routines that intercede when the system abnormally ends your server program. System code X'3C6' in *z/OS MVS System Codes* describes the IARSUBSP macro errors that cause your program to abnormally end.

To set up a recovery routine for any program, you must understand the topics presented in the recovery section in *z/OS MVS Programming: Authorized Assembler Services Guide*. To design recovery for programs running in subspaces, you need additional information about the recovery routine's **subspace environment**. The subspace environment is simply whether the routine is running with full address space addressability or in a subspace, and, if it is running in a subspace, which one? Like a mainline program, a recovery or retry routine can use the BSG instruction to:

- Change subspaces, by specifying the ALET of the desired subspace
- Run with full address space addressability, by specifying ALET 0
- Return to the last subspace to have control, by specifying ALET 1.

(See “Using special alets” on page 223 for a more information on using ALETs 0 and 1 with the BSG instruction.) Given that, consider these questions:

- In what environment does the recovery routine receive control?
- Does a recovery routine that changes its environment need to ensure that the environment is reset if the recovery routine abnormally ends?
- In what environment does a retry routine receive control?

These questions are answered in the following topics.

Planning for SPIE and ESPIE routines

SPIE and ESPIE exit routines and data areas should reside in storage that can be referenced by all subspaces. This ensures that a SPIE or ESPIE routine has addressability to all required data areas, regardless of the subspace environment in which the program interruption occurs.

Because SPIE and ESPIE routines cannot percolate, they always receive control in the subspace environment that was in effect when the error occurred in the mainline program. SPIE and ESPIE routines are explained in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Planning for ESTAE-type recovery routines and FRRs

The remaining information on planning for recovery applies to both ESTAE-type recovery routines and FRRs, unless otherwise noted.

Subspace environment at entry to recovery routines

A recovery routine runs in the subspace environment that the previous routine was running in when it encountered an error or percolated.

After an error in the mainline program, the first recovery routine to receive control runs in the subspace that the mainline was running in at the time of error. If the mainline routine was running in Subspace A, the recovery routine gets control in Subspace A. If the mainline routine was running with full address space addressability, the recovery routine gets control with full address space addressability.

If the recovery routine percolates, the next recovery routine receives control in the environment in effect when the previous recovery routine percolated. For example, if the first recovery routine received control in Subspace A, issued the BSG instruction to change to full address space addressability, then percolated, the next recovery routine will receive control with full address space addressability.

Resetting a changed subspace environment after a recovery routine error

A recovery routine that abnormally ends might cause the next recovery routine to get control in the wrong subspace. The SETRP macro with SSRESET=YES requests that the system reset the environment when an ESTAE-type recovery routine abnormally ends. SSRESET cannot be used by FRRs. and has no effect when a recovery routine percolates.

When the current recovery routine temporarily changes subspaces, specify SSRESET=YES to protect the next recovery routine. SSRESET=YES ensures that, if the current recovery routine abnormally ends before it returns to the correct subspace, the next recovery routine will get control in the subspace in which the current routine received control. This allows you to ensure that the next recovery routine receives control in the correct subspace, regardless of the subspace the current routine runs in when it abnormally ends.

When the current recovery routine processes successfully and returns to the correct subspace, SSRESET=YES protection is no longer necessary. The next recovery routine is no longer in danger of receiving control in the wrong subspace if the current recovery routine abnormally ends. At this point, you can specify SSRESET=NO in the current recovery routine. SSRESET=NO negates the earlier specification of SSRESET=YES. If the current recovery routine abnormally ends after specifying SSRESET=NO, the next recovery routine gets control as described in “Subspace environment at entry to recovery routines” on page 231. See *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for a description of the SETRP macro and the SSRESET parameter.

Passing information to a recovery routine in a subspace

When the system supplies an SDWA, the system provides it in storage that can be referenced by all subspaces. A recovery routine running in a subspace does not have to do anything extra to reference the SDWA.

If the mainline program passes a user parameter area to the recovery routine, the mainline routine should create the user parameter area in storage that can be referenced by all subspaces. This ensures that, if

the recovery routine changes the environment in which it is running, it will still be able to reference the user parameter area.

Subspace environment on entry to retry routine

A retry routine gets control in the subspace environment in which the last recovery routine returned to RTM. If the last recovery routine was running in a different subspace environment from the mainline program, the recovery routine should issue the BSG instruction to ensure that the mainline resumes processing in the correct subspace environment.

Diagnosing errors in a subspace environment

The following diagnostic information is available for programs that use subspaces.

Diagnosing 0C4 abends

A program running in a subspace cannot reference storage that is subspace-eligible but not assigned to the program's subspace. If a program attempts to reference this storage, the program will incur either a page translation exception or a segment translation exception, and the program will abnormally end with an X'0C4' system completion code. This abend occurs when the subspace-eligible storage is not assigned to the program's subspace at the time of error.

Using IPCS to diagnose program errors in a subspace

The following IPCS subcommands can help you diagnose errors in an address space that owns subspaces:

- The RSMDATA subcommand allows you to produce a subspace report.
- The STATUS subcommand with the FAILDATA option includes the subspace environment at the time of error.
- The NAME subcommand displays the subspace name and address space identifier when the STOKEN specified is a subspace STOKEN.
- The SUMMARY subcommand with the FORMAT keyword generates a report that indicates whether a program was running in a subspace when the error occurred.
- The VERBEXIT subcommand with the LOGDATA verb name formats LOGREC buffer records that indicate whether a program was running in a subspace when the error occurred and, if so, include the subspace name and STOKEN.

See *z/OS MVS IPCS Commands* for details.

RSM component trace

RSM component trace provides options that allow you to trace subspace services. See *z/OS MVS Diagnosis: Tools and Service Aids* for details.

Requesting a dump

If a program requests an SVC dump while running in a subspace, the system dumps the entire address space.

If a program requests an ABEND dump while running in a subspace (by specifying a SYSABEND, SYSMDUMP, or SYSUDUMP DD statement in the job step), the system dumps only the storage that the program can reference while running in a subspace.

Appendix A. Accessibility

Accessible publications for this product are offered through IBM Documentation (www.ibm.com/docs/en/zos).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact the z/OS team web page (www.ibm.com/systems/campaignmail/z/zos/contact_z) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Documentation with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1)

are mutually exclusive alternatives. If you hear the lines `3.1 USERID` and `3.1 SYSTEMID`, your syntax can include either `USERID` or `SYSTEMID`, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The `*` symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element `*FILE` with dotted decimal number 3 is given the format `3 \* FILE`. Format `3* FILE` indicates that syntax element `FILE` repeats. Format `3* \* FILE` indicates that syntax element `* FILE` repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines `5.1*, 5.1 LASTRUN, and 5.1 DELETE` mean that if you use more than one of the `LASTRUN` and `DELETE` syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line `2.1 %OP1` means that you must refer to separate syntax fragment `OP1`.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example `5? NOTIFY`). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines `5 ?, 5 NOTIFY`, and `5 UPDATE`, you know that the syntax elements `NOTIFY` and `UPDATE` are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the `!` symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines `2? FILE, 2.1! (KEEP), and 2.1 (DELETE)`, you know that `(KEEP)` is the default option for the `FILE` keyword. In the example, if you include the `FILE` keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the `FILE` keyword is omitted, the default `FILE(KEEP)` is used. However, if you hear the lines `2? FILE, 2.1, 2.1.1! (KEEP)`, and `2.1.1 (DELETE)`, the default option KEEP applies only to the next higher dotted decimal number, `2.1` (which does not have an associated keyword), and does not apply to `2? FILE`. Nothing is used if the keyword `FILE` is omitted.

*** indicates an optional syntax element that is repeatable**
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data area`, you know that you can include one data area, more than one data area, or no data area. If you hear the lines `3* , 3 HOST, 3 STATE`, you know that you can include `HOST, STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you 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
Site Counsel
2455 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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMSdfp, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those

products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming Interface Information

This book is intended to help the customer to code programs with needs that extend beyond the boundaries of the address space in which the programs are dispatched. This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Glossary

This glossary includes definitions of terms used in this book.

Access list
A table in which each entry specifies an address space, data space, or hiperspace that a program can reference.

Address space
A range of two gigabytes of contiguous virtual storage addresses that the system creates for a user. It contains user data and programs, as well as system data and programs, some of which are common to all address spaces. Instructions execute in an address space.

Address/data space
Address space or data space

ALET (access list entry token)
A token that indexes into an access list. When a program is in AR mode and the ALET is in an AR with the corresponding GPR being used as a base register, the ALET indicates the address space or data space that the system is to reference. An ALET can also identify a hiperspace.

ALET-qualified address
An 8-byte address that consists of an ALET (identifying a space) and an address (identifying an offset within the space). The ALET-qualified address might be contained in storage with the ALET in the first 4 bytes and the address in the last 4 bytes. Or, it might be contained in an AR/GPR pair.

AR (access register) mode
The ASC mode in which the system uses both the GPR (used as the base register) and the corresponding AR to resolve an address in an address/data space.

AR/GPR
Access register and general purpose register pair.

ASC (address space control) mode
The mode (determined by the PSW) that tells the system where to find the data it is to reference. Three ASC modes are AR, secondary, and primary.

Authority table
Consists of entries that define the PT and SSAR authority that other address spaces have with respect to the cross memory user's address space. Entries also define the EAX authority that a PC routine has with respect to an address space. Each entry in the table corresponds to a particular authorization index.

Authorization index (AX)
Indicates the authority of a program that's running in one address space to issue the PT or SSAR instruction with another address space as the target of those instructions. A program runs with the AX of the current primary address space.

Auxiliary storage
Data storage that is not central or expanded storage, typically, storage on direct access devices (DASDs).

BAKR (branch and stack) instruction
Branches to a location and adds an entry to the linkage stack.

bar
A virtual line that marks the 2-gigabyte address in a 64-bit address space. It separates virtual storage below the 2-gigabyte address (called “below the bar”) from virtual storage above the 2-gigabyte line (called “above the bar”).

Basic PC instruction
Transfers control to another program, the PC routine. The basic PC requires the service provider to save and restore the user's environment. The PC routine can be in the same address space as the program that issues the PC instruction, or in a different address space.

Central storage
Program-addressable storage from which instructions and other data can be loaded directly into registers for subsequent execution or processing.

Control parameters
Parameters that a macro service routine uses.

CPYA (copy access) instruction
Copies the contents of one AR into another AR.

Cross memory mode
Cross memory mode exists when at least one of the following conditions are true:

- The current primary address space (PASN) and the current home address space (HASN) are different address spaces.
- The current secondary address space (SASN) and the current home address space (HASN) are different address spaces.
- The ASC mode is secondary.

Data space
A range of up to two gigabytes of contiguous virtual storage addresses that a program can directly manipulate through assembler instructions. Unlike an address space, a data space can hold only data; it does not contain common areas or system data or programs. Instructions do not execute in a data space.

Dispatchable unit
A TCB or SRB, sometimes called work unit.

DU-AL (dispatchable unit access list)
Access list associated with the dispatchable unit (or work unit).

EAR (extract access) instruction
Copies the contents of an AR into a GPR.

EAX (extended authorization index)
An identifier that the system uses to determine the authority of a program to add entries to or delete entries from its access lists and use ARs to access data in address spaces.

Entry table
A table in which each entry defines the attributes of a PC routine.

EPAR (extract primary ASN) instruction
Place the primary ASID into a GPR.

EREG (extract stacked registers) instruction
Loads ARs and GPRs from the current linkage stack entry.

ESAR (extract secondary ASN) instruction
Place the secondary ASID into a GPR.

ESO (expanded storage only) hiperspace
A hiperspace that is backed by expanded storage only. It is a high-speed buffer area or "cache" for storing data.

ESTA (extract stacked state) instruction
Obtains non-register information from the current linkage stack entry.

Expanded storage
High-speed high-volume electronic storage. The operating system transfers this storage to and from central storage in 4K byte blocks.

GPR
General purpose register

Guard area
An area of a memory object that a program cannot access. The guard area is optionally established when the IARV64 macro creates a memory object; a guard area can be changed into an accessible area through the IARV64 macro.

Hiperspace
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 can hold user data; it does not contain common areas or system data. Instructions do not execute in a hiperspace. Unlike an address space or a data space, data is not directly addressable. To manipulate data in a hiperspace, you bring the data into the address space in 4K byte blocks.

Home address space
The address space in which the TCB or the SRB is initially dispatched. A TCB or SRB represents a dispatchable work unit. When MVS initially dispatches a work unit the primary, secondary, and home address spaces are the same address space.

IAC (insert address space control) instruction
Indicates in a GPR which ASC mode is in effect.

LAE (load address extended) instruction
Load a specified ALET and address into an AR/GPR pair.

LAM (load access multiple) instruction
Load the contents of one or more ARs from a specified location.

Linkage index (LX)
Provides an index into the cross memory user's linkage table.

Linkage stack
An area that the system provides for a BAKR or a stacking PC to save status information. It can be either a normal linkage stack or a recovery linkage stack.

Linkage table
A table in which each entry points to an entry table. It provides the connection between a PC number and an entry table.

MEMLIMIT
The limit on the use of virtual storage above the bar. You can set this limit through JCL EXEC statements, JCL JOB statements, SMF commands, the SMFPRMxx parmlib member, and the IEFUSI installation exit. The system default is 2G.

Memory object
An area of virtual storage that a program creates using the IARV64 macro; it resides above the 2-gigabyte address in a 64-bit address space.

MSTA (modify stacked state) instruction
Copies the contents of an even/odd GPR pair to the modifiable area of the current linkage stack entry.

MVCK (move with key) instruction
Moves data between storage areas that have different protection keys.

MVCP (move to primary) instruction
Moves data from the secondary address space to the primary address space.

MVCS (move to secondary) instruction
Moves data from the primary address space to the secondary address space.

Non-shared standard hiperspace
A standard hiperspace that can generally be shared only with programs that are dispatched in the owner's home address space. However, a program not dispatched in the owner's home address space and using an ALET, can access this non-shared standard hiperspace through the owner's home PASN-AL.

Normal linkage stack
Saves program status information. The system uses this linkage stack until entries are no longer available. It then presents a "stack-full" program interruption.

PR (program return) instruction
Returns control to a program that issued a stacking PC or BAKR instruction.

Primary address space
The address space whose segment table is used to fetch instructions in primary, secondary, and AR ASC modes. A program in primary mode fetches data from the primary address space.

Primary ASC mode
The ASC mode in which the system uses the GPRs, but not the ARs, to resolve an address in an address space. In primary ASC mode, the system fetches instructions and data from the primary address space.

Processor storage
The combination of central and expanded storage.

Program call (PC) instruction
Transfers control to another program, the PC routine. The PC instruction is either a stacking or a basic PC instruction. The PC routine can be in the same address space as the program that issues the PC instruction, or, in the case of a space-switching PC instruction, a different address space.

Program call (PC) routine
A program that receives control as the result of a PC instruction's executing and performs a service for the caller.

PT (program transfer) instruction
Returns control to a program that issued a basic PC instruction.

Recovery linkage stack
The linkage stack that is available to the program's recovery routines after the "stack full" interruption occurs on a normal linkage stack.

SAC (set address space control) instruction
Explicitly sets either the primary ASC mode, secondary ASC mode, AR ASC mode, or home ASC mode.

SAR (set access register) instruction
Place the contents of a GPR into an AR.

Secondary address space
The address space whose segment table the system uses to access data in secondary ASC mode.

Secondary ASC mode
The ASC mode in which the system fetches instructions from the primary address space and data from the secondary address space.

Shared standard hiperspace
A standard hiperspace that can be shared with programs that are dispatched in any address space.

Space-switching PC instruction
A PC instruction that transfers control to a PC routine that is not in the same address space as the program that issues the PC instruction.

SSAR (set secondary ASN)
Sets the secondary address space to a specified address space.

Stacking PC instruction
Transfers control to another program, the PC routine. The stacking PC uses the linkage stack for storing the caller's status. It provides more options and more automatic function than the basic PC instruction. The PC routine can be in the same address space as the program that issues the PC instruction, or a different address space.

Stack-full program interruption
A program interruption that occurs when a program tries to add an entry to a linkage stack and no more entries are available.

STAM (store access register multiple) instruction
Store the contents of one or more ARs beginning at a specified location.

Standard hiperspace
A hiperspace that is backed by expanded storage, and auxiliary storage, if necessary. Through a buffer area in an address space, your program can "scroll" through a standard hiperspace.

STOKEN (space token)
An eight-byte identifier of an address space, data space, or hiperspace. It is similar to an address space identifier (ASID or ASN), with two important differences: the system does not reuse the STOKEN value within an IPL, and data spaces do not have ASIDs. Macros that support AR mode callers often use STOKENs instead of ASIDs to identify address/data spaces.

TTOKEN
A 16-byte identifier of a TCB. Unlike a TCB address, a TTOKEN is unique within an IPL.

User parameters
Parameters that a macro service routine passes from the user of the macro to another routine.

Work unit
A TCB or SRB, sometimes called a dispatchable unit.

X-macro
Some macro services offer two macros, one for callers in primary mode and one for callers in AR mode. The name of the macro for the AR mode caller is the same as the name of the macro for primary mode callers, except the macro that supports the AR mode caller ends with an "X". The "X" version of the macro is called an "X-macro".

64-bit address space
The address space that is supported by an architecture that provides 64-bit addresses.

Index

Numerics

0C4 system completion code
  with subspaces 232
0D3 system completion code
  with subspaces 223
12A system completion code
  with subspaces 220
3C6 system completion code
  IARSUBSP ASSIGN request 224
  IARSUBSP IDENTIFY request 222
  IARSUBSP UNASSIGN request 227
  IARSUBSP UNIDENTIFY request 228
  recovering from 230
64-bit address space
  using assembler instructions
    binary operations 68
  what is 59
64-bit addressing mode (AMODE)
  modal instructions
    AMODE 24 70
    AMODE 31 70
    AMODE 64 70
  non-modal instructions 69
64-bit instructions
  pitfalls to avoid 71

A

A05 system completion code
  with subspaces 228
A0A system completion code
  with subspaces 228
A78 system completion code
  with subspaces 228
ABEND macro used to dump data space storage 156
access data in a data space
  rules for 129, 131
access list
  adding entry for address space 107
  adding entry for data space 107
  adding entry for hiperspace 175
  definition 95
  deleting hiperspace entry 176
  description 98
  illustration 127
  private entries 102
  public entries 102
  relationship with work unit 100
  size 100
  type 98, 102, 128
access list entry
  adding 98, 106
  deleting 113, 114, 176
  example 107
  limit 157
  type 102
access list entry token 223
ACCESS parameter on ALESERV macro 106, 115
accessibility
  contact IBM 233
  features 233
add an entry for primary address space to DU-AL 112
add an entry to an access list
  description 106
  example 107, 108, 175
add an entry to the DU-AL
  rules for data spaces 131
add entry to an access list
  rules for data spaces 131
add entry to the DU-AL
  example 109, 144
  illustration 109
add entry to the PASN-AL
  example 110
  illustration 110
  rules for data spaces 131
ADD parameter on ALESERV macro 105
ADDPASN parameter on ALESERV macro 105
address space
  comparison with data space and hiperspace 1
  creating through ASCRE 197
  creating through ASCRE macro
    example 207
  deleting through ASDES macro 207
  establishing access through ARs 98
  establishing attributes for 206
  getting EAX-authority 115
  naming 197
  procedures for obtaining EAX-authority 118
  terminating with ASDES macro 197
address space authorization for PC routines
  purpose 27
address space creation
  initialization routine 200
  synchronization of process 201
  through ASCRE 197
address/data space
  definition 94
addressability
  changing
    with subspaces 225
  establishing
    to subspace 223
  full address space 211
addressability through DU-AL
  example 109
  illustration 109
addressability through PASN-AL
  example 110
  illustration 110
addressing mode
  compared to ASC mode 94
AKM (authorized key mask)

AKM (authorized key mask) *(continued)*
purpose 27
relationship to PKM 27
AL parameter on ALESERV macro 106
ALCOPY parameter on ATTACHX macro 147
ALESERV macro
ADD request
example 107, 109, 110, 122, 144, 147, 149, 154, 155, 175
process for SCOPE=COMMON data space 145
use 105, 106, 108
ADDPASN request
use 105, 108, 112
DELETE request
example 114, 122, 145, 150, 155, 156, 176
use 105
EXTRACT request
use 105, 108
EXTRACTH request
use 105, 108
SEARCH request
use 105, 106
summary of functions 105
ALET (access list entry token)
checking validity 120
definition 95
definition of special 103
example of loading a zero into an AR 104
example of loading into AR 98
for a hiperspace 174
for home address space 103, 113
for primary address space 103
for secondary address space 103
illustration 95, 127
illustration of special 103
obtaining 108
obtaining for hiperspace 175
passing 108, 175
passing across address spaces 111
reuse 114, 146
rules for passing 108
special 98, 103, 223
with a value of 0 103
with a value of 1 103
with a value of 2 103
ALET parameter on ALESERV macro 106
ALET-qualified address
definition 96
used in macro parameter list 123
AR (access register)
advantage 93
compared with cross memory 93
contents 95
description 93
example of loading ALET 98
example of loading an ALET of zero 104
rules for coding 96
used for EAX-authority 115
using for data reference 93
why a program would use 93, 94
AR information
formatting and displaying 125
AR instruction
summary 97
AR mode
coding instructions 96
compared to primary mode 94, 95
definition 94
description 3, 4, 94
importance of comma 97
importance of the contents of ARs 104
issuing macros 123
passing parameters 125
rules for coding 96
switching 94
AR mode data movement
compared to cross memory data movement 117
ARCHECK subcommand
format and display AR information 125
ARR (associated recovery routine)
identifying 48
ASC (address space control) mode
description 94
ASC mode
compared to addressing mode 94
compared to cross memory mode 94
description 3
switching 94
ASCRE macro
cross memory environment 204
establishing attributes 206
establishing cross memory environment 202
establishing termination routine 205
use 197
ASDES macro
description 197
example 208
use 207
ASEXT macro
description 197
use 204
ASID (address space identifier)
compared with an STOKEN 98
illustration of reuse 54
reuse of 53
assembler instructions
changing modifiable area of linkage stack entry 14
manipulating entries on linkage stack 12
modify ARs 97
used for cross memory 22
assign
ownership of data space 128
assistive technologies 233
associated recovery routine 48
asynchronous communication
definition 3
asynchronous exit routine
associated DU-AL 123
associated EAX 123
associated PASN-AL 123
AT (authority table)
illustration 201
PT authority 27
purpose 27
relationship to AX 27
SSAR authority 27
ATSET macro
example 40, 42

ATSET macro *(continued)*
purpose 21
used for obtaining EAX 119
used to obtain EAX-authority 120
attach a subtask and pass a DU-AL 147
ATTACH macro
used to pass DU-AL to subtask 147
ATTACHX macro
example of passing DU-AL to subtask 147
used to pass DU-AL to subtask 147
ATTR parameter on ASCRE 206
authority
to set up addressability to address spaces 106
to set up addressability to data spaces 105
authorization index 19
authorization key mask 19
AX (authorization index)
compared with an EAX 117
illustration 117, 201
reuse of 57
value 27
AXEXT macro
purpose 21
AXFRE macro
example 42
purpose 21
AXLIST parameter on ASCRE macro 203
AXRES macro
example 36
purpose 21
used to get EAX-authority 119
AXSET macro
example
resetting an AX 42, 44
setting an AX 36, 43
purpose 21

B

BAKR instruction
adding entry to linkage stack 12
description 12
example 13, 154, 155
base space 211
basic decision
data space or hiperspace 5
basic PC
available to all address spaces 32
available to selected address spaces 33
overview 23
PC routine execution 23
BLOCKS parameter on DSPSERV macro 133, 151, 165, 166, 192
BSG instruction
with special ALETs 223

C

callable cell pool service
for data space 141
CALLERKEY parameter on DSPSERV macro 133, 165
CALLRTM macro
terminating address space created by ASCRE macro 207
CASTOUT parameter on DSPSERV macro 168
change
EAX value 120
characteristics of access lists 99
check
ALET of caller 120
EAX-authority of caller 120
global bit for AR mode 122
check validity of ALET
example 120
checkpoint/restart
managing data space storage 138
managing hiperspace storage 160
checkpointing
with subspaces 226
CHKEAX parameter on ALESERV macro 105, 115
choose the name of a data space 134
comma
careful use of in AR mode 97
common area data space 128
comparison of a PASN-AL and a DU-AL 100
comparison of EAX and AX 117
contact
z/OS 233
contents of an AR 95
contents of linkage stack 14
CPYA instruction
description 97
create
address space 197
data space 127
ESO hiperspace 169
hiperspace 159
standard hiperspace 168
cross memory
example of setting up 197
setting up environment through ASCRE macro 197
cross memory communication
accessing data from a PC routine 24
accounting considerations 31
advantage 19
assembler instructions used for 22
basic PC linkage, overview 23
considerations before using 30
entry table 26
environment 26
environmental considerations 31
establishing communication 31
EX (entry table index) 27
example 35, 45
execution time consideration 31
introduction 19
linkage conventions 49
linkage table 26
macros used for 21
overview of cross memory communication 22
PC linkage 23, 45
PC number 26
PC routine
characteristics 45
execution 23
invocation 23
overview 22
requirements 48

cross memory communication *(continued)*
PKM (PSW– key mask) 27
recovery considerations 57
requirements 31
requirements for PC routines 48
resource management considerations 31
restrictions 31
services for all address spaces 32
services for selected address spaces 33
stacking PC linkage, overview 23
summary 25
terminology 19
when to use 19
cross memory data movement
compared to AR mode data movement 117
cross memory environment
example 203
illustration 202
cross memory mode
compared to AR mode 93
compared to ASC mode 94
cross memory recommendations
general register 13 initialization 51
loading PC routines 57
macro sequence 32
obtaining and releasing resources 52
type of PC to use 26
use of ETDEF macro 37
use of IHAETD mapping macro 37
use of PCLINK macro 49
current entry in linkage stack
definition 12
current size of data space 135
current size of hiperspace 166
CVT (communications vector table)
testing
for subspace 218

D

data movement
in AR mode 117
data privacy 75
data reference
using ARs 93
data space
choosing the name 134
compared to address space 1
compared with hiperspace 6
containing DREF storage 137
creating 127, 133
data manipulation 6
data manipulation illustration 6
decision to use 5
definition 127
deleting 129, 144
description 4
dumps of storage 156
efficient use 157
establishing access through ARs 98
example 5, 8
example of creating 107
example of moving data in and out 139
extending current size 129, 143
data space *(continued)*
identifying the origin 136
managing storage 138
mapping data-in-virtual object into 5, 148
physical backing 7
protecting storage 137
PSW key 137
referencing data 127
releasing storage 151
restoring after a checkpoint/restart operation 138
saving before a checkpoint/restart operation 138
shared between two address spaces 111
storage available for 139
summary of rules 131
unmapping data-in-virtual object into 150
use 5
use by SRB 153
use of physical storage 7
data space and hiperspace
comparing 163
data space or hiperspace
which one should you use 8
data space storage
dumping 156
extending 131
managing 138
physical backing 7
protecting 137
releasing 131, 151
rules for releasing 151
serializing use 139
data-in-virtual
mapping a hiperspace object to an address space window 195
mapping into a data space 4, 5, 148
mapping into a hiperspace 5, 192, 193
data-only space
definition 1
illustration 1
delete
access list entry
example 114, 145, 176
address space through ASDES macro 207
data space
description 144
example 114, 144, 145
rules 129, 131
hiperspace
description 191
example 176, 196
DELETE parameter on ALESERV macro 105
deletion
of subspace 228
of subspace entry 227
diagnosis
of subspace errors 232
difference
between data spaces and hiperspaces 6
dispatchable unit access list 223
displaying AR information 125
DIV macro
example 149, 195
mapping a data-in-virtual object to a hiperspace
example 194

DIV macro *(continued)*
mapping a hiperspace as a data-in-virtual object
example 195
use 4, 5, 148, 192
DREF parameter on DSPSERV macro 133
DREF storage in data space
defining 137
definition 137
DSPSERV macro
CREATE request
example 107, 122, 136, 137, 144, 147, 149, 154, 155, 166, 168, 169, 175, 178, 194, 195
example of use by SRB 154, 155
use 147
creating DREF storage 137
DELETE request
example 114, 144, 145, 150, 155, 156, 176, 191, 195, 196
EXTEND request
example 143, 191
LOAD option
use 150
OUT option
use 150
RELEASE request
use 151, 191
DSPSTOR parameter on SNAPX macro 156
DU-AL
adding subspace entry 223
associated with asynchronous exit routine 123
characteristic 99
compared to PASN-AL 99, 100
containing subspaces
copying 223
definition 98, 128
description 99
illustration of a space switch 101
illustration of accessing data space 130
illustration of PASN-AL and DU-AL 100
DUMPOPX parameter on ABEND macro 156

E

EAEASWT ECB 201, 202
EAERIMWT ECB 201, 202
EAR instruction
description 97
EAX (extended authorization index)
associated with asynchronous exit routine 123
changing 120
compared with an AX 117
definition 102, 115, 117
description 117
freeing 120
illustration 117
reserving 119
reuse of 57
unauthorized 115
with the value 0 115
EAX-authority
checked by system 106
checking 120
compared with SSAR authority 119
definition 102, 115
EAX-authority *(continued)*
description 115
illustration 116
obtaining 115
procedures for obtaining 119
system checking for 116
EAX-checking
how to prevent it 115
how to request it 115
ECBs for initialization routine 201, 202
EKM (entry key mask)
purpose 27
relationship to PKM 27
entry table
connecting, example 40, 44
example 44
example of how to define 37
illustration 201
ownership 37
purpose 26
purpose of EX 26
structure 26
entry table index 19
EREG instruction
description 14
example 14
ESO hiperspace
backing 168
compared with standard hiperspace 163
creating 168
definition 161
description 162
example of creating 169
read and write operation 179
use 162
ESTA instruction
description 14, 15
example 15, 16
establish
access for ARs 98
cross memory environment
through ASCRE macro 197, 202
establish addressability
example 108
to a data space
definition 95, 127
example 138, 144
procedures 138
rules 129, 131
to an address space
definition 95
establish attributes for address spaces 206
ESTAE-type recovery routine
use of linkage stack 16
ET (entry table)
illustration 201
ETCON macro
example 40, 44
purpose 21
used to obtain EAX-authority 119
ETCRE macro
example 37, 44
purpose 21
used for obtaining EAX 119

ETDEF macro
example 37
purpose 21
used to change EAX 119
ETDES macro
example 42, 44
purpose 21
ETDIS macro
example 42
purpose 21
EX (entry table index)
purpose 27
responsibility for maintaining 27
example of moving data in and out of data space 139
examples of cross memory usage
provide services to all address spaces
address space authorization 43
cleaning up 44
establishing access 44
granting PT authority 43
granting SSAR authority 43
providing service 44
removing access 44
setting up 43
system LX, obtaining 43
providing non-space switch service 45
providing services to selected address spaces
constructing a PC number 39
entry table create 37
establishing access 40
granting PT authority 40
granting SSAR authority 40
PC routine definition 37
removing access to PC routine 42
reserving an AX 36
reserving an EAX 36
reserving an LX 36
execution key mask 19
extend current size of data space
example 143
procedure 143
rules 129, 131
extend current size of hiperspace
example 191
procedure 190
EXTEND parameter on DSPSERV macro 143, 190
extended addressability
basic concepts 3
introduction 1
EXTRACT parameter on ALESERV macro 105

F

FAILDATA subcommand 232
feedback xvii
formatting AR information 125
FPROT parameter on DSPSERV macro 133, 137, 165, 167
free
EAX value 120
full address space addressability 211

G

GENNAME parameter on DSPSERV macro 133, 134, 164, 165
glossary of terms 241
GPR/AR
definition 94
illustration 94
guard area
changing its size 88

H

hiperspace
as data-in-virtual object 195
choosing the name 165
compared to address space 1
compared with data space 6
creating 159, 164
data manipulation 7
decision to use 5
definition 1, 159
deleting 191
deleting hiperspace from access list 176
description 4
efficient data transfer 180
example 8
example of creating 175
extending current size 190
identifying the origin 168
illustration 1
managing storage 160
manipulating data
illustration 159
mapping data-in-virtual object into 5, 192, 193
obtaining an ALET 174
physical backing 7
problem state program using 170
protecting storage 167
PSW key 167
referencing data 176
releasing storage 191
requesting amount of storage 166
restoring after a checkpoint/restart operation 160
rules for problem state programs 170, 172
rules for supervisor state programs 172
saving before a checkpoint/restart operation 160
storage available 160
summary of rules 163
transferring data to and from address space 176
type 161
use by SRB 196
use of physical storage 7
hiperspace or data space
which one should you use 8
hiperspace storage
managing 160
physical backing 8
protecting 167
releasing 191
rules for releasing 192
serializing use 161
home address space
ALET for 103, 113

HSPALET parameter on HSPSERV macro 180
HSPSERV macro
  compared to IOSADMF macro 180
  CREAD and CWRITE operation
    example 180
  faster data transfer 180
  read operation 176, 177, 179
  SREAD and SWRITE operation
    example 178
    illustration 176
  write operation 176
HSTYPE parameter on DSPSERV macro 164

I

IARSUBSP macro
  ASSIGN parameter 223
  CREATE parameter 222
  DELETE parameter 228
  IDENTIFY parameter 220
  RANGLIST parameter 221
  UNASSIGN parameter 227
  UNIDENTIFY parameter 228
IARV64 services
  use 75
identify the origin of the data space 136
IEANTCR callable service
  example of using 40
IEANTRT callable service
  example of using 41
IEFUSI installation exit 139, 160
IEZEAECB mapping macro 198, 201
IHAASEO mapping macro 198
IHAETD mapping macro 37
information field in linkage stack entry
  definition 15
  illustration 15
initial size of data space 135
initial size of hiperspace 166
initialization routine for new address space
  description 200
  how to write 200
  requirement 198
  specifying 197
installation limit
  amount of storage for data space and hiperspace 134
  on amount of storage for data space and hiperspace 139, 160, 166
  on size of data space 139
  on size of hiperspace 166
  on size of hiperspaces 160
  size of data space 134
instructions used for cross memory 22
instructions used to manipulate linkage stack entry 14
IOSADMF macro
  APURGE request 184
  AREAD and AWRITE request 183
  compared to HSPSERV macro 180
  efficient data transfer 183
  example of 184
IPCS (interactive problem control system)
  format and display AR information 125

K

KEEP parameter on HSPSERV macro 180
KEY parameter on DSPSERV macro 133, 165, 167
keyboard
  navigation 233
  PF keys 233
  shortcut keys 233

L

LAE instruction
  description 97
  example 154, 155
LAM instruction
  description 97
  example 98, 107, 144, 175
large pages 67
limit use of data space 139
limit use of hiperspace 160
linkage conventions 71
linkage index 19
linkage stack
  adding entry 12
  advantages of using 3, 11
  assembler instructions that manipulate entries 12
  default number of entries 16
  description 11
  dumping the contents 17
  example 13
  expanding 16
  format of information field 15
  illustration 11
  removing entry 12
  use by ESTAE-type recovery routine 16
  use by reentrant programs 11
linkage stack entry
  assembler instructions that manipulate 12
  contents 14
linkage stack instructions
  using 14
linkage table
  illustration 201
  purpose 26
  relationship to LX 26
LISTD parameter on SDUMPX macro 156
load instruction in AR mode
  example 97
LSEXPAND macro
  example 16
  use 16
LX
  extended non-system example 37
  reusable extended non-system example 37
  reuse of 56
LX (linkage index)
  owner 36
LX reuse facility 20
LXFRE macro
  example 42
  purpose 21
LXLIST parameter on ASCRE macro 203
LXRES macro
  example 36, 43

LXRES macro *(continued)*
purpose 21
used for obtaining EAX 119

M

macros
cross memory
requirements for issuing 32
summary 21
issuing in AR mode 123
passing parameters to in AR mode 125
manage data space storage 138
manipulate data in a data space 144
manipulate data in hiperspace 159
map data-in-virtual object into data space
rules for problem state programs 148
rules for supervisor state programs 149
map data-in-virtual object into hiperspace
example 194
rules for problem state programs 192
rules for supervisor state programs 193
map hiperspace as data-in-virtual object
example 195
mapping macros
IEZEAECB mapping macro 198, 201
maximum size of data space 135
maximum size of hiperspace 166
MEMLIMIT
definition 59
determining 63
memory management
above the bar 61
memory object
attributes 61
common 61
discard data 88
example of creating with a guard area 90
IARV64 list request 91
large pages 67
limiting use 62
ownership 80
pagefix 62
protecting storage 75
releasing physical resources that back pages of 88
using 71
memory objects
data privacy 75
mode
AR 94
ASC mode 94
primary 94
modifiable area in linkage stack
changing 14
MSTA instruction
description 14, 16
example 16
use 16
MVC instruction
example in AR mode 95
example in primary mode 95
MVCP instruction
compared to MVC in AR mode 117
MVCS instruction
compared to MVC in AR mode 117
MVCS instruction *(continued)*
compared to MVC in AR mode 117
MVS macros
issuing in AR mode 123
passing parameters to in AR mode 125

N

name a data space 134
name a hiperspace 165
NAME parameter on DSPSERV macro 133, 134, 164, 165
name/token callable services
example of using 35
navigation
keyboard 233
non-shared standard hiperspace
definition 162
non-space switch PC routine
definition 23
normal linkage stack
definition 11
NUMRANGE parameter on HSPSERV macro 177

O

obtain
ALET for the primary address space
illustration 113
EAX-authority
procedures for 118
storage in another address space 122
origin of data space 136
origin of hiperspace 168
ORIGIN parameter on DSPSERV macro 136, 168
OUTNAME parameter on DSPSERV 165
OUTNAME parameter on DSPSERV macro 133, 134
ownership of data space
assigning to another TCB 128
definition 128
ownership of hiperspace
assigning to a TCB 196
definition 161

P

page data space pages into central storage
rules 129
using DSPSERV LOAD and OUT 150
pages, large 67
parameter
passing in AR mode 125
parameters
passing through ASCRE macro 205
passing to new address space 197
receiving through ASEXT macro 205
PASN-AL
associated with asynchronous exit routine 123
characteristic 99
compared to DU-AL 99, 100
definition 98, 128
description 99
illustration of a space switch 101
illustration of accessing data space 130

PASN-AL *(continued)*
illustration of PASN-AL and DU-AL 100
pass ALET
to MVS macros
rules for 125
pass ALETs
across address spaces
illustration 110, 111
rules for 108
pass DU-AL to subtask 147
pass STOKENs to another program
illustration 111
passing ALETs
to other programs
rules for 108
PC linkage
overview 23
type 23
PC number
construction example 39
example of how to provide 35
how to construct 26
purpose 26
PC routine
accessing data 24
authorization for problem state routines 27
available to all address spaces
address space authorization 32
AX value used 32
basic PC routine linkage 33
entry table connect 33
entry table create 33
linkage index 33
macros used 32
PC routine 33
PT authority 32
SSAR authority 32
available to selected address spaces
address space authorization 34
authorization index 34
AX value 34
entry table 34
linkage 34
linkage index 34
macros used 33
PC number 35
PC routine 34, 35
PT authority 34
SSAR authority 34
stacking AX value 34
stacking PC,address space authorization 35
basic
addressing mode 47
authorization for problem state programs 46
defined 23
defining 45
linkage capability 45
linkage conventions 49
non-space switch 46
PKM (PSW-key mask) 47
problem state 46
requirements 48
space switch 46
supervisor state 46
PC routine *(continued)*
comparison of linkage conventions 52
defining 45
definitions, common to basic and stacking 46
execution 23
IBM recommendation 45
invocation overview 23
invocation, example 41
linkage capability 45
linkage conventions
basic PC 49
stacking PC 50
loading recommendations 57
MVCP instruction, using 24
MVCS instruction, using 24
non-space switch, defined 23
overview 22
requirements 48
space switch, defined 23
stacking
addressing mode 47
ARR (associated recovery routine) 48
ASC mode 47
authorization for problem state programs 46
defined 23
defining 45
EAX (extended authorization index) 47
linkage capability 45
non-space switch 46
PKM (PSW-key mask) 47
problem state 46
PSW key 48
requirements 48
SASN value 48
space switch 46
supervisor state 46
use of access registers (ARs) 24
used in obtaining EAX-authority 119
PCLINK macro
compared with linkage stack function 11
purpose 21
physical storage
comparison of data space and hiperspace use 7
PKM (PSW key mask)
purpose with PC routine 27
relationship to AKM 27
relationship to EKM 27
PPT values of new address space 197
PR instruction
description 12
example 13
removing entry from linkage stack 12
primary address space
adding an entry to DU-AL 112
ALET for 103
primary mode
compared to AR mode 95
compared with AR mode 94
definition 94
description 3, 94
switching 94
privacy, memory object data 75
private entry in access list
compared to public entry 116

private entry in access list *(continued)*
definition 102, 115
illustration 116
private memory object
creating
example 77
creating, using and freeing a
example 78
fixing pages
example 87
fixing the pages of 87
freeing
example 78
problem state program
use of data spaces and hiperspaces 6
program note
for using SCOPE=COMMON data space 146
protect data space storage 137
protect hiperspace storage
illustration 167
protection
of data
in a subspace 211
PSW key
protecting data space storage 137
protecting hiperspace storage 167
PSW key mask 19
PT authority
definition 23
public entry in access list
compared to private entry 116
definition 102, 115
illustration 116

R

range list
description 221
error 222, 224, 227, 228
illustration 221
requirements
ASSIGN request 224
IDENTIFY request 221
UNASSIGN request 224
UNIDENTIFY request 221
RANGLIST parameter on HSPSERV macro 177, 179
read from an ESO hiperspace 179
read operation
for ESO hiperspace 179
for standard hiperspace 176, 177
recovery
in subspace
ESPIE routine 231
ESTAE-type routine 231
FRR 231
SPIE routine 231
recovery considerations for cross memory 57
recovery linkage stack
definition 12
reentrant programs use of linkage stack 11
relationship between data space and owner 128
relationship between linkage stack and ESTAE-type recovery routine 16
release
release *(continued)*
data space storage
rules for 151
hiperspace storage
rules for 192
RELEASE parameter on HSPSERV macro 177
remove
entry from access list 114, 176
requirements
cross memory 31
reserve
EAX 119
resetting
subspace environment 231
resource management in a cross memory environment
accounting considerations 57
PC routines 57
restarting
with subspaces 226
restrictions
cross memory 31
RSMDATA subcommand 232
rules for
passing ALET
to MVS macros 125
passing ALETs
to other programs 108
running in subspace 211

S

SAC instruction
example 107, 122, 154, 155, 175
SAR instruction
description 97
example 122
SCOPE parameter on DSPSERV macro 128, 133
SCOPE=ALL data space
definition 128
illustration of accessing 130
use 128
SCOPE=COMMON data space
compared with CSA 145
creating and using 145
definition 128
illustration of using 146
use 128, 145
SCOPE=SINGLE data space
definition 128
illustration of accessing 130
use 128
SDUMPX macro used to dump data space storage 156
SEARCH parameter on ALESERV macro 105
secondary address space
ALET for 103
sending to IBM
reader comments xvii
serialize use
data space storage 139
hiperspace storage 161
set
ASC mode through SAC instruction 94
set up
addressability to a data space

set up *(continued)*
addressability to a data space *(continued)*
example 107
addressability to a hiperspace
example 175
addressability to a subspace 223
addressability to an address space 98, 105
cross memory environment in new address space 204
set up EAX-authority to an address space 115
SETLOCK macro
example 154, 155
use 154, 155
SETRP macro
SSRESET parameter 231
share data spaces
between two address spaces 111
shared data space
between two problem state programs 147
shared memory object
accessing
example 83
changing
example 83
creating
example 82
freeing
example 83
shared standard hiperspace
definition 162
shortcut keys 233
size of data space
specifying 134
size of hiperspace
specifying 166
SMF installation exit IEFUSI 139, 160
SNAPX macro used to dump data space storage 156
space switch PC routine
definition 23
space-switching PC instruction
affect on addressability through access lists 101
special ALETs
adding entry to the DU-AL 98
definition 103
illustration 103
passing to other programs 108
SRB (service request block)
example of using data space 155
use 3
use of data space 151
use of hiperspace 196
SSAR authority
compared with EAX-authority 119
SSRESET parameter
of SETRP macro 231
stacking PC
adding entry to linkage stack 12
available to all address spaces 32
available to selected address spaces 33
overview 23
PC routine execution 23
STAM instruction
description 97
standard hiperspace
compared with ESO hiperspace 163
standard hiperspace *(continued)*
creating 168
definition 161
description 161
example of creating 168
example of scrolling 161
examples of use by problem state programs 170
illustration of scrolling 161
read and write operation 177
use 162
START parameter on DSPSERV macro 151, 192
STOKEN
definition 127
returned by DSPSERV macro 127
STOKEN (space taken)
obtaining from DSPSERV 108
obtaining from other programs 108
passing to another program 108
STOKEN (space token)
compared with an ASID 98
definition 98
illustration of passing to another program 109
passing to another program 111
STOKEN parameter on ALESERV macro 106
STOKEN parameter on ASCRE macro 106
STOKEN parameter on DIV macro 148
STOKEN parameter on DSPSERV macro 106, 133, 164
STOKEN parameter on HSPSERV macro 177
storage
alignment
for subspace 220
assigning to subspace 223
attributes
for subspace 219
backing
for subspace 219
eligible to be assigned to subspace 211
isolation
within address space 211
making eligible for subspace 220
managing data space 138, 141
managing hiperspace 160
obtaining
for subspace 219
obtaining storage in another address space 122
referenced by all subspaces 211
releasing after subspace 228
required by system
with subspaces 215
storage available for data space 139
storage available for hiperspace 160
STORAGE macro
OBTAIN request
example 122, 179
RELEASE request
example 122
use 122
subspace
assigning storage 223
benefits 214
creation 222
deletion 228
description 211
establishing addressability 223

subspace *(continued)*
identifying storage 220
limitations 214
making storage ineligible 228
obtaining storage 219
resetting 231
running 225
storage
attributes 219
system storage overhead 215
testing CVT 218
using MVS services 227
subspace-eligible storage 211
SUMLSTL parameter on SDUMPX macro 156
summary of cross memory communication 25
synchronous communication
definition 3
synchronous cross memory communication 19
SYSSTATE macro
example 122, 124
use 123
system linkage index
purpose 43
saving 43

T

TCBTOKEN macro
TYPE parameter 154
use 154, 155
using to find TTOKEN 154
terminating address space with ASDES macro 197
termination routine for new address space 205, 207
terminology 241
TESTART macro
use 120
testing
for subspace 218
TKLIST parameter on ASCRE macro 203
trademarks 240
TRMEXIT parameter on ASCRE macro 205
TTOKEN parameter on DSPSERV 155
TTOKEN parameter on DSPSERV macro
example 154, 155
example of using data space 154

U

unmap a data-in-virtual object 150
use of the ALET for home address space
illustration 113
use the ALET for home address space
example 113
user interface
ISPF 233
TSO/E 233
UTOKEN parameter on ASCRE macro 205

V

VIO (virtual input/output)
comparison with data space and hiperspace 9
virtual storage
virtual storage *(continued)*
why use above the bar
use, example 61
VLF (virtual lookaside facility)
use 4

W

work unit
definition 98
relationship to access list 100
write operation
for ESO hiperspace 179
for standard hiperspace 176
write to a standard hiperspace 177
write to an ESO hiperspace 179

X

X-macro
definition 124
rules for using 124

Z

z/Architecture
setting and checking the addressing mode 70
z/Architecture instructions
using the 64-bit GPR 69
z/Architecture processes S/390 instructions, how
examples 68

Product Number: 5650-ZOS

SA23-1394-40