IBM z/VSE
5.1

Extended Addressability

SC34-2630-02

Note: Before using this information and the product it supports, be sure to read the general information under "Notices" on page 149.

This edition applies to Version 5 Release 1 of IBM® z/Virtual Storage Extended (z/VSE), Program Number 5609-ZV5, and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC34–2630–01.

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

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

```
IBM Deutschland Research & Development GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany
```

You may also send your comments by FAX or via the Internet:

```
Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456
```

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

Contents

Figures

Tables

About This Book

This manual describes the support summarized as extended addressability and which is available with Version 5 Release 1 of IBM z/VSE. z/VSE belongs to the z/Architecture® operating systems designed for the z/Architecture environment.

Who Should Use This Book

The manual is intended mainly for those who plan and write programs and applications for a z/VSE customer environment. A knowledge of z/VSE and assembler programming is required.

How to Use This Book

The information in this manual is divided into five parts:

Part 1. 31-Bit Addressing Support
Part 2. 64-Bit Addressing Support
Part 3. Data Spaces and Virtual Disks
Part 4. Programming Enhancements
Part 5. Appendixes

Part 4 provides additional usage information for the topics discussed in Part 1 and Part 2.

Where to Find More Information

Related manuals are cited in the text where appropriate. In general, the information provided in this manual is closely related to the information provided in the following z/Architecture Principles of Operation IBM manual.

z/VSE IBM Documentation

IBM Documentation is the new home for IBM's technical information. The z/VSE IBM Documentation can be found here:

https://www.ibm.com/docs/en/zvse/6.2

You can also find VSE user examples (in zipped format) at

https://public.dhe.ibm.com/eserver/zseries/zos/vse/pdf3/zVSE_Samples.pdf

Summary of Changes

These are the enhancements that have been made available via the *June 2013* Service Upgrade of z/VSE 5.1:

- The support for I/O operations on memory objects is new. See "Using 64-Bit Virtual I/O Operations on Memory Objects" on page 53.

These are the new items and changes that were delivered *at General Availability of z/VSE V5R1:*

- A section has been introduced that describes how you can use the *64-bit address space* via *memory objects* to obtain additional virtual storage. See Chapter 7, "Using the 64-Bit Address Space," on page 39.
- The table contained in "z/VSE Macros and Their Mode Dependencies" on page 123 now includes AMODE64 information related to the use of the 64-bit address space and memory objects.
- Other minor changes and improvements have been included in this manual.

Part 1. 31-Bit Addressing Support

1. The information and examples provided for 31-bit addressing in this manual are based on functions of the High Level Assembler.
2. High Level Assembler refers to *High Level Assembler Version 1.6 for z/OS, z/VM, and z/VSE*, which is a base program of Version 5 Release 1 of IBM z/VSE.

Chapter 1. Introducing 24-Bit / 31-Bit AMODE and RMODE

This topic provides an introduction to 24-bit and 31-bit program *addressing modes*.

A program can have one of the following addressing modes:

- 24-bit addressing mode (a *24-bit program*). Specified using the AMODE 24 and RMODE 24 program attributes.
- 31-bit addressing mode (a *31-bit program*). Specified using the AMODE 31 and RMODE ANY program attributes.

The use of AMODE and RMODE to specify *24-bit and 31-bit addressing modes* are discussed in the remainder of this topic.

Note: z/VSE does not support AMODE and RMODE attributes for *64-bit addressing*. For details of the 64-bit addressing mode, see Chapter 7, “Using the 64-Bit Address Space,” on page 39.

General Considerations for AMODE and RMODE

To determine whether a program is to run below 16 MB or above 16 MB, z/VSE analyses the program attributes AMODE and RMODE assigned to a program. AMODE and RMODE are the programmer's specification of the addressing mode in which a program is expected to get control and where a program is expected to reside in virtual storage.

AMODE (Addressing Mode)

AMODE is a program attribute that can be specified or is assigned as default for each CSECT or phase. It refers to the address length a program is prepared to handle when it gets control. For AMODE, you can specify one of the following values:

AMODE 24
The program is designed to receive control in 24-bit addressing mode. In this mode, the processor treats all virtual addresses as 24-bit values.

AMODE 31
The program is designed to receive control in 31-bit addressing mode. In this mode, the processor treats all virtual addresses as 31-bit values.

AMODE ANY
The program is designed to receive control in either 24-bit or 31-bit addressing mode. The final decision about 24-bit or 31-bit addressing mode is open until the program receives control.

RMODE (Residency Mode)

RMODE is a program attribute that can be specified (or is assigned as default) for each CSECT or phase. RMODE states the virtual storage location (either below 16 MB or anywhere in virtual storage) where the program is expected to reside. For RMODE, you can specify one of the following values:

RMODE 24
The program is designed to reside below 16 MB in virtual storage.

RMODE ANY
The program is designed to reside at any virtual storage location, either above or below 16 MB but always *below the 2 GB bar*.

Specifying AMODE and RMODE

Programmers can specify AMODE and RMODE for new programs but also for old programs through:

1. Reassembly or recompilation.

   Addressing mode and residency mode can be specified in the High Level Assembler or VS COBOL II.
2. Using the linkage editor MODE control statement or PARM values in the EXEC LNKEDT statement.

z/VSE assigns default attributes to any program that does not have AMODE and RMODE specified.

AMODE and RMODE Combinations at Program-Run Time

When the program gets control at execution time, there are **only three** valid AMODE/RMODE combinations:

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

Programming Aspects

AMODE

A program's AMODE attribute determines whether the program is to receive control in 24-bit or 31-bit addressing mode. Once a program gets control, the program can change the AMODE if necessary.

In 24-bit addressing mode, the processor treats all virtual addresses as 24-bit values. This makes it impossible for a program in 24-bit addressing mode to address virtual storage with an address greater than 16,777,215 (16 MB) because that is the largest number a 24-bit binary field can hold. In 31-bit addressing mode, the processor treats all virtual addresses as 31-bit values.

The ability of a processor to permit the execution of programs in 24-bit addressing mode as well as programs in 31-bit addressing mode is referred to as **bimodal operation**.

Processors which support bimodal operation ensure that both, new programs and most old programs, can execute correctly. Bimodal operation is necessary because certain coding practices in existing programs depend on 24-bit addresses. For example:

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

Each phase has an **AMODE** attribute. A CSECT can have only one AMODE, which applies to all its entry points. Different CSECTs of a phase can have different AMODEs.

RMODE

Each phase has an **RMODE** attribute. **RMODE** specifies where a program is expected to reside in virtual storage:

- RMODE 24 indicates that a program is coded to reside in virtual storage below 16 MB.
- RMODE ANY indicates that a program is coded to reside anywhere in virtual storage - either above or below 16 MB but always *below the 2 GB bar*.

Programs that Must Reside Below 16 MB

The following types of programs must reside below 16 MB (addressable by 24-bit callers):

- Programs that have the AMODE 24 attribute
- Programs that have the AMODE ANY attribute
- Programs that use system services that require their callers to be AMODE 24
- Programs that use system services that require their callers to be RMODE 24
- Programs that must be addressable by callers with AMODE 24.
- Programs that use 2-byte or 3-byte relocatable address constants.

Programs without these characteristics can reside anywhere in virtual storage.

Rules and Conventions for 31-Bit Addressing

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

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

   Any data passed from a 31-bit addressing mode program to a 24-bit addressing mode program must reside in virtual storage below 16 MB (A 24-bit addressing mode program cannot reference data above 16 MB without changing addressing mode).
2. The A-mode bit affects the way some instructions work.

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

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

Changing the AMODE

To change addressing mode it is necessary to change the value of the PSW A-mode bit. This can be done in one of the following ways:

- By using the mode setting instructions BASSM and BSM.
- By using the z/VSE AMODESW macro (there is *no* AMODE 64 support).
- By using the addressing mode setting instructions SAM24, SAM31 and SAM64.

Refer also to “How to Change the AMODE” on page 19.

Mode Sensitive Instructions

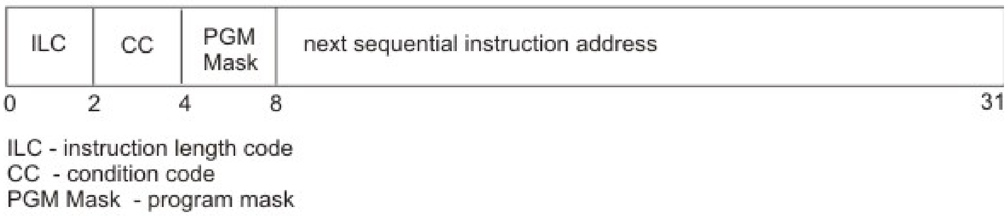
The processor is sensitive to the addressing mode that is in effect (the setting of the PSW AMODE bit). The current PSW controls instruction sequencing. The instruction address field in the current PSW contains either a 24-bit address, 31-bit address, or 64-bit address depending on the current setting of the PSW AMODE bits (bits 31 and 32). For those instructions that develop or use addresses, the addressing mode in effect in the current PSW determines whether the addresses are 24, 31, or 64 bits long.

The *z/Architecture Principles of Operation* manual provide a complete description of the instructions available. The following topics provide an overview of mode sensitive and branching instructions regarding the 24-bit and 31-bit addressing modes. The 64-bit addressing mode is described in Chapter 7, “Using the 64-Bit Address Space,” on page 39.

BAL and BALR

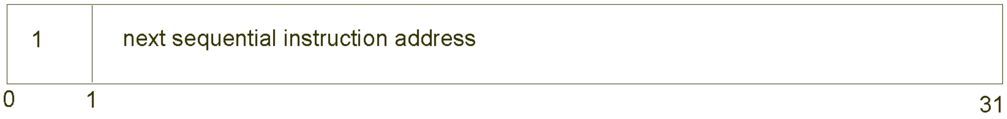
BAL and BALR are addressing-mode sensitive. In 24-bit addressing mode, BAL and BALR put link information into the high-order byte of the first operand register and put the return address into the remaining three bytes before branching.

First operand register (24-bit addressing mode)


ILC
CC
PGM Mask
next sequential instruction address
0
2
4
8
31
ILC - instruction length code
CC - condition code
PGM Mask - program mask

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

First operand register (31-bit addressing mode)



When executing in 31-bit addressing mode, BAL and BALR do not save the instruction length code, the condition code, or the program mask.

LA

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

LRA

When executed in the 24-bit or 31-bit addressing mode, the LRA (load real address) instruction always results in a 31-bit real address. The virtual address specified is based on the value of the PSW A-mode bits (bits 31 and 32) at the time the LRA instruction is executed.

AMODE Processing Capabilities

BASSM and BSM

BASSM (branch and save and set mode) and BSM (branch and set mode) are branching instructions that manipulate the PSW A-mode bits (bits 31 and 32). Programs can use BASSM when branching to modules that might have different addressing modes. Programs invoked through a BASSM instruction can use a BSM instruction to return in the caller's addressing mode. BASSM and BSM are described in more detail in Chapter 4, "Establishing Linkage in a 31-Bit Addressing Environment," on page 21.

BAS and BASR

BAS and BASR (branch and save) are branching instructions which

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

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

SAM24 and SAM31

The SAM24 and SAM31 (Set Addressing Mode) instructions only set AMODE 24 and AMODE 31. A previous AMODE setting is not saved.

AMODESW Macro

With the AMODESW macro, a program can switch addressing modes.

For a summary of the macro's functions refer to "AMODESW Macro" on page 128.

Note on the CALL macro:

The CALL macro does not change the AMODE; that is, the AMODE of the caller is passed to the called program.

z/VSE System Services and 31-Bit Addressing

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

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

Other services have restrictions with respect to address parameter values and might require one or more parameter addresses to be below 16 MB. Some of these services accept callers to be in AMODE 24 or AMODE 31, whereas others require callers to be in AMODE 24.

Some services do not support 31-bit addressing. Refer to "z/VSE Macros and Their Mode Dependencies" on page 123 for details.

Chapter 2. Planning for 31-Bit Programs

This topic describes how you can convert your programs from 24-bit addressing mode to 31-bit addressing mode.

A program can have one of the following *addressing modes*:

- 24-bit addressing mode (a *24-bit program*), specified using the AMODE 24 and RMODE 24 program attributes.
- 31-bit addressing mode (a *31-bit program*), specified using the AMODE 31 and RMODE ANY program attributes.
- 64-bit addressing mode (a *64-bit program*), specified explicitly using the IARV64 macro. Such programs can create or maintain memory objects in the 64-bit addressing area (above the 2 GB bar).

The *64-bit addressing mode* is described in Chapter 7, "Using the 64-Bit Address Space," on page 39.

Most user programs that are running on earlier VSE/SP and VSE/ESA systems will also run unchanged in AMODE 24 on z/VSE. It is recommended that the partition in which such a program is running does not cross the 16 MB line.

Some programs need to be modified to execute in AMODE 31 and provide the same functions. Still other programs need to be modified to run in AMODE 24.

The following sections helps you determine what changes to make to a program you want to convert to AMODE 31 and what is to consider when writing new 31-bit code.

Some reasons for converting to AMODE 31 are:

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

Converting Existing Programs

Keeping in mind that AMODE 31 programs can reside either below or above 16 MB (or cross the 16 MB line), you can convert existing programs as described below.

Converting a Program to Use 31-Bit Addresses

This requires a change in AMODE only:

- You can change the entire module to use 31-bit addressing.
- You can change only that portion that requires 31-bit addressing mode execution.

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

At best, converting a program into 31-bit mode means just another linkage editor run.

Figure 1 on page 10 summarizes the actions required to maintain the proper interface with a program you plan to change to AMODE 31.

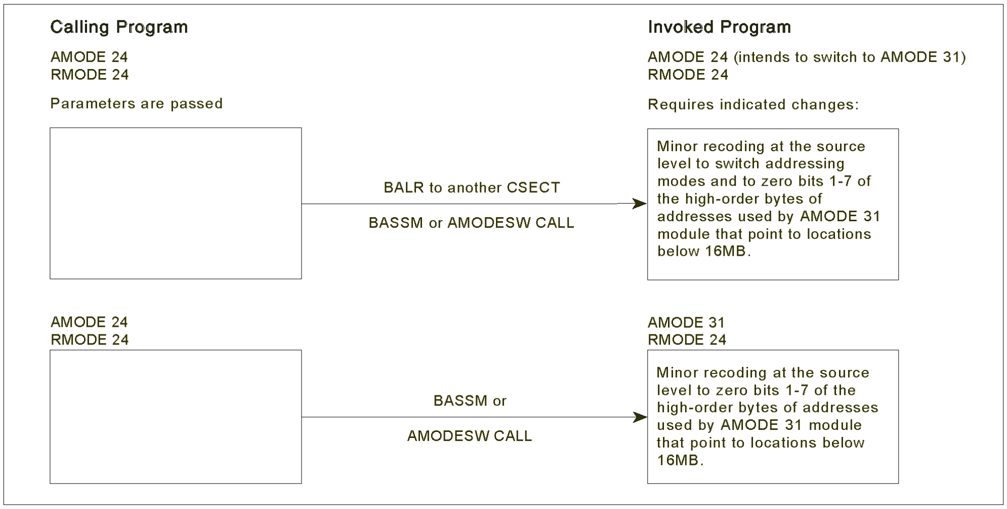

Calling Program
AMODE 24
RMODE 24
Parameters are passed
Invoked Program
AMODE 24 (intends to switch to AMODE 31)
RMODE 24
Requires indicated changes:
BALR to another CSECT
BASSM or AMODESW CALL
Minor recoding at the source level to switch addressing modes and to zero bits 1-7 of the high-order bytes of addresses used by AMODE 31 module that point to locations below 16MB.
AMODE 24
RMODE 24
AMODE 31
RMODE 24
BASSM or
AMODESW CALL
Minor recoding at the source level to zero bits 1-7 of the high-order bytes of addresses used by AMODE 31 module that point to locations below 16MB.

Figure 1. Maintaining Correct Interfaces to Programs

Moving a Program above 16 MB

This requires a change in both AMODE and RMODE.

In general, you move an existing program above 16 MB because there is not enough space below 16 MB. For example:

- An existing program or application is growing so large that it no longer fits below 16 MB.
- An existing application that now runs as a series of separate programs, or that executes in an overlay structure, would be easier to manage as one large program.
- Code is in the SVA (24-Bit) and moving it to the SVA (31-Bit) would provide more space for the private area below 16 MB.

The techniques that are used to establish proper interfaces to modules that move above 16 MB depend on the number of callers and the ways they invoke the program. Table 1 on page 10 summarizes the techniques for passing control. The programs involved must ensure that any addresses passed as parameters are treated correctly (high-order bytes of addresses to be used by a AMODE 31 program must be validated or zeroed).

Table 1. Establishing Correct Interfaces to Programs that Move Above 16 MB

Means of Entry to Moved Module (AMODE 31,RMODE ANY)	Few AMODE 24,RMODE 24 Callers	Many AMODE 24,RMODE 24 Callers
• **BALR** **or** • **Macro LOAD and BASSM instruction** **or** • **Macro CDLOAD and BASSM instruction**	• Have caller use macros LOAD/ CDLOAD and BASSM instruction (invoked program returns via BSM instruction) or • Change caller to AMODE 31, RMODE 24 before performing call.	Create a linkage assist routine (described in Chapter 4, "Establishing Linkage in a 31-Bit Addressing Environment," on page 21). Give the linkage assist routine the name of the moved program.

Table 1. Establishing Correct Interfaces to Programs that Move Above 16 MB (continued)

Means of Entry to Moved Module (AMODE 31,RMODE ANY)	**Few AMODE 24,RMODE 24 Callers**	**Many AMODE 24,RMODE 24 Callers**
BALR using an address in a common control block	• Have caller switch to AMODE 31 via BSM instruction or • Change the address in the control block to a pointer-defined value (described in Chapter 4, “Establishing Linkage in a 31-Bit Addressing Environment,” on page 21) and use BASSM instruction. The moved program uses instruction BSM to return.	Create a linkage assist routine (described in Chapter 4, “Establishing Linkage in a 31-Bit Addressing Environment,” on page 21).

Note: BASSM and BSM instructions can be replaced by the AMODESW macro. For an example, refer to “How to Change the AMODE” on page 19.

In deciding whether or not to modify a program to execute in AMODE 31 either below or above 16 MB, there are several considerations:

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

Among the specific practices to check for are:

1. Does the module depend on the instruction length code, condition code, or program mask placed in the high order byte of the return address register by a 24-bit mode BAL or BALR instruction? One way to determine some of the dependencies is by checking all uses of the SPM (set program mask) instruction. SPM might indicate places where BAL or BALR were used to save the old program mask, which SPM might then have reset. The IPM (insert program mask) instruction can be used to save the condition code and the program mask.
2. Does the module use an LA instruction to clear the high-order byte of a register? This practice will not clear the high-order byte in AMODE 31.
3. Are any address fields that are less than 4 bytes still appropriate? Make sure that a load instruction does not pick up a 4-byte field containing a 3-byte address with extraneous data in the high-order byte. Make sure that bits 1-7 are zero.
4. Does the program use the ICM (insert characters under mask) instruction? The use of this instruction is sometimes a problem because it can put data into the high-order byte of a register containing an address, or it can put a 3-byte address into a register without first zeroing the register. If the register is then used as a base, index, or branch address register in AMODE 31, it might not contain the proper address.
5. Does the program invoke AMODE 24 programs? If so, shared data must be below 16 MB.
6. Is the program invoked by AMODE 24 or 31 programs? Is the data in an area addressable by the programs that need to use it? The data must be below 16 MB if used by an AMODE 24 program.

Writing New Programs that Use 31-Bit Addresses

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

If your program resides above 16 MB, it needs a separate phase to perform the linkage to an unchanged AMODE 24 program or service. Such phases are called linkage assist routines and are described under Chapter 4, “Establishing Linkage in a 31-Bit Addressing Environment,” on page 21.

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

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

  ```
  Rather than:    L    1,A

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

  The 7 specifies a 4-bit mask of 0111. The ICM instruction shown inserts bytes beginning at location A+1 into register 1 under control of the mask. The bytes to be filled correspond to the 1 bits in the mask. Because the high-order byte in register 1 corresponds to the 0 bit in the mask, it is not filled.
- If the program needs storage above 16 MB, obtain the storage by using the GETVIS macro with LOC=ANY. **This is the only form that allows you to obtain storage above 16 MB**. Do not use storage areas above 16 MB for system save areas (subtask save areas, for example) and, possibly, parameters that need to be passed to other programs.
- To make debugging easier, switch addressing modes only when necessary.
- Identify the intended AMODE and RMODE for the program in a prologue.
- User-written STXIT routines need to be aware of the restricted support of the BC mode PSW fields in the new exit routine save area. Refer to “STXIT Macro” on page 134 for details.
- The CALL macro cannot be used to switch the AMODE.

When writing new programs, you need to decide whether to use AMODE 24 or AMODE 31.

For AMODE 24 you must write, for example, service routines that use system services requiring entry in AMODE 24 or that must accept control directly from AMODE 24 programs.

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

New Programs below 16 MB

The main reason for writing new AMODE 31 programs which reside below 16 MB is to be able to address areas above 16 MB or to invoke AMODE 31 programs while, at the same time, simplifying communication with existing AMODE 24 programs or system services.

Even though your program resides below 16 MB, you must be concerned about dealing with programs that require entry in AMODE 24 or that require parameters to be below 16 MB. Figure 6 on page 22 in Chapter 4, “Establishing Linkage in a 31-Bit Addressing Environment,” on page 21 contains more information about parameter requirements.

New Programs above 16 MB

When you write new programs that reside above 16 MB, your main concerns are:

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

Writing 31-Bit Programs for a Mixed z/VSE Environment

Up to VSE/ESA 1.2, VSE/ESA supported 24-bit programs only. Starting with VSE/ESA 1.3, VSE/ESA and later z/VSE support 31-bit addressing. This means that for a mixed environment special considerations are necessary as outlined below.

Programs designed to execute on systems with either 24 or 31-bit addressing mode must use fullword addresses where possible and use no new functions. Programs must also be aware of **downward incompatible** macros. Some of these can be made downward compatible by using the SPLEVEL macro. Refer to “SPLEVEL Macro” on page 133. If this is not possible, the old (downward level) macro library must be used for compile or assembly.

Dual Programs

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

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

Using the SPLEVEL Macro

There exist macros where the level of the macro expansion generated during an assembly depends on the value of an assembler language global SET symbol. If the SET symbol value is 1, the system generates VSE/ESA 1.1/1.2 expansions; if it is 2 or 3, the system generates VSE/ESA 1.3 expansions; and if it is 4, the system generates expansions suitable for VSE/ESA 2.1 and later.

The SPLEVEL macro allows programmers to change the value of the SET symbol. The SPLEVEL macro shipped with VSE/ESA 2.1 sets a default value of 4 for the SET symbol. Therefore, unless a program or installation specifically changes the default value, the macros generated are VSE/ESA 2.1 macro expansions.

The SPLEVEL macro sets the SET symbol value for that program's assembly only and affects only the expansions within the program being assembled. A single program can include multiple SPLEVEL macros to generate different macro expansions. The example in Figure 2 on page 14 shows how to obtain different macro expansions within the same program and make a test at execution time to determine which expansion to execute.

```
*   DETERMINE WHICH SYSTEM IS EXECUTING
         .
         .
*   PREPARE INPUT REGISTER
         SUBSID INQUIRY,NAME=SUP,AREA=(2),LEN=(3)
*   CHECK WHETHER PROGRAM IS RUNNING ON VSE/ESA 1.1/1.2,
*   VSE/ESA 1.3, or VSE/ESA 2.1
         USING   XYZ,2
         CLI     IJBSVERS,X'06'
         BNL     SP4
         CLI     IJBSVERS,X'05'
         BL      SP5
         CLI     IJBSREL,X'02'
         BNL     SP3
         DROP    2
*   INVOKE THE VSE/ESA 1.1/1.2 VERSION OF THE PFIX MACRO
SP1      SPLEVEL SET=1
         PFIX    .....
         B       CONTINUE
SP3      EQU     *
*   INVOKE THE VSE/ESA 1.3 VERSION OF THE PFIX MACRO
         SPLEVEL SET=3
         PFIX    ......
         B       CONTINUE
SP4      EQU     *
*   INVOKE THE VSE/ESA 2.1 VERSION OF THE PFIX MACRO
         SPLEVEL SET=4
         PFIX    ......
         B       CONTINUE
SP5      ......            VSE/SP NOT CONSIDERED
CONTINUE EQU     *
         .
         .
SUP      DC      C'SUP '
         .
         .
XYZ      MAPSSID
```

Figure 2. Example of How to Use the SPLEVEL Macro

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

Chapter 3. Using AMODE and RMODE to Specify 24-Bit / 31-Bit Addressing Modes

This topic describes how you can use AMODE and RMODE to specify 24-bit and 31-bit *addressing modes*.

A program can have one of the following addressing modes:

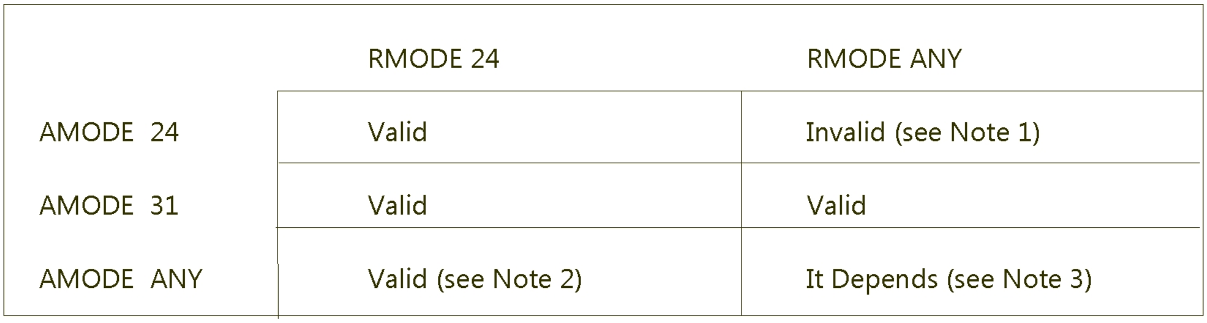
- 24-bit addressing mode (a *24-bit program*), specified using the AMODE 24 and RMODE 24 program attributes.
- 31-bit addressing mode (a *31-bit program*), specified using the AMODE 31 and RMODE ANY program attributes.

For an overview of how AMODE and RMODE are used, refer to Chapter 1, "Introducing 24-Bit / 31-Bit AMODE and RMODE," on page 3.

Note: z/VSE does not support AMODE and RMODE attributes for *64-bit addressing*. For details of the 64-bit addressing mode, see Chapter 7, "Using the 64-Bit Address Space," on page 39.

AMODE and RMODE Combinations

Figure 3 on page 15 shows all possible AMODE and RMODE combinations and indicates which are valid.



	RMODE 24	RMODE ANY
AMODE 24	Valid	Invalid (see Note 1)
AMODE 31	Valid	Valid
AMODE ANY	Valid (see Note 2)	It Depends (see Note 3)

Figure 3. Possible AMODE and RMODE Combinations

Note:

1. This combination is invalid because an AMODE 24 module cannot reside above 16 MB.
2. This is a valid combination in that the assembler and linkage editor accept it from all sources. However, the combination is not used at execution time. Specifying ANY is a way of deferring a decision about the actual AMODE until the last possible moment before execution. At execution time, the module must execute in either AMODE 24 or AMODE 31.
3. The attributes AMODE ANY/RMODE ANY take on a special meaning when used together (this meaning might seem to disagree with the meaning of either taken alone). A program with the AMODE ANY/RMODE ANY attributes will execute on either a z/VSE system that does support 31-bit addresses or on a z/VSE system that does not support 31-bit addresses if the program is designed to:

   - Use no facilities that are unique to 31-bit addressing.
   - Execute entirely in AMODE 31 on a system that supports 31-bit addresses and returns control to its caller in AMODE 31 (the AMODE could be different from invocation to invocation).
   - Execute entirely in 24-bit addressing mode on a system without 31-bit addressing.

   The linkage editor accepts this combination from the object module but not from the PARM field of the linkage editor EXEC statement or the linkage editor MODE control statement. Refer also to "AMODE/RMODE Combinations from the ESD" on page 119.

AMODE and RMODE Combinations at Execution Time

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

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

Determining the AMODE and RMODE of a Phase

There are various ways to find out the AMODE and RMODE assigned to a phase:

- You can look at the source code (AMODE/RMODE statement of the High Level Assembler) to determine the AMODE and RMODE intended for the program. However, the linkage editor can override these specifications.
- You can look at the linkage editor map which lists the AMODE and RMODE of the phase and of each CSECT included in the phase. Refer to z/VSE Diagnosis Tools for a description of the linkage editor map.
- You can create a librarian LISTDIR printout or display which shows the AMODE and RMODE of the phases stored in a sublibrary.
- You can use the CDLOAD and LOAD macros as described in the manual z/VSE System Macros Reference.

High Level Assembler Support of AMODE and RMODE

The High Level Assembler supports AMODE and RMODE assembler instructions. By using such instructions, you can specify an AMODE and an RMODE to be associated with a control section, an unnamed control section, or a named common control section.

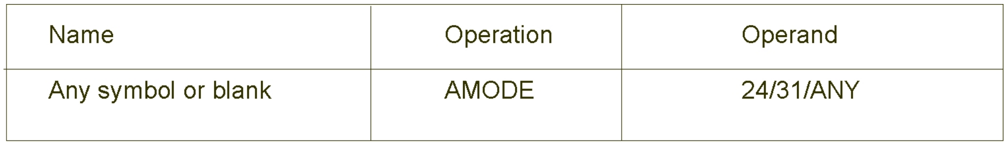
AMODE and RMODE in the Object Module

The assembler also checks for the following error conditions:

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

AMODE and RMODE Assembler Instructions

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



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

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

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

Name	Operation	Operand
Any symbol or blank	RMODE	24/ANY

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

The High Level Assembler selects the following defaults when AMODE, RMODE, or both are not specified:

Specified
: **Defaulted**

Neither
: AMODE 24 RMODE 24

AMODE 24
: RMODE 24

AMODE 31
: RMODE 24

AMODE ANY
: RMODE 24

RMODE 24
: AMODE 24

RMODE ANY
: AMODE 31

Linkage Editor Support of AMODE and RMODE

Note: Under Appendix A, “Linkage Editor and Librarian Support,” on page 117 you find additional background information about the 31-bit support provided by the linkage editor.

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

- ESD (external symbol dictionary) entries in object modules.
- PARM field of the linkage editor EXEC statement. For example:

  ```
  // EXEC LNKEDT,PARM='AMODE=31,RMODE=ANY,.....'
  ```

  PARM field input overrides object module input.
- Linkage editor MODE control statements. For example:

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

  MODE control statement input overrides object module and PARM field input.

Linkage editor processing results in AMODE and RMODE indicators located in the library directory entry for the phase linked.

The linkage editor creates AMODE/RMODE indicators in the library directory entry based not only on input from the object module but also on the PARM field of the linkage editor EXEC statement and the MODE control statements. The last two sources of input override indicators from the object module. Figure 4 on page 18 shows linkage editor processing of AMODE and RMODE.

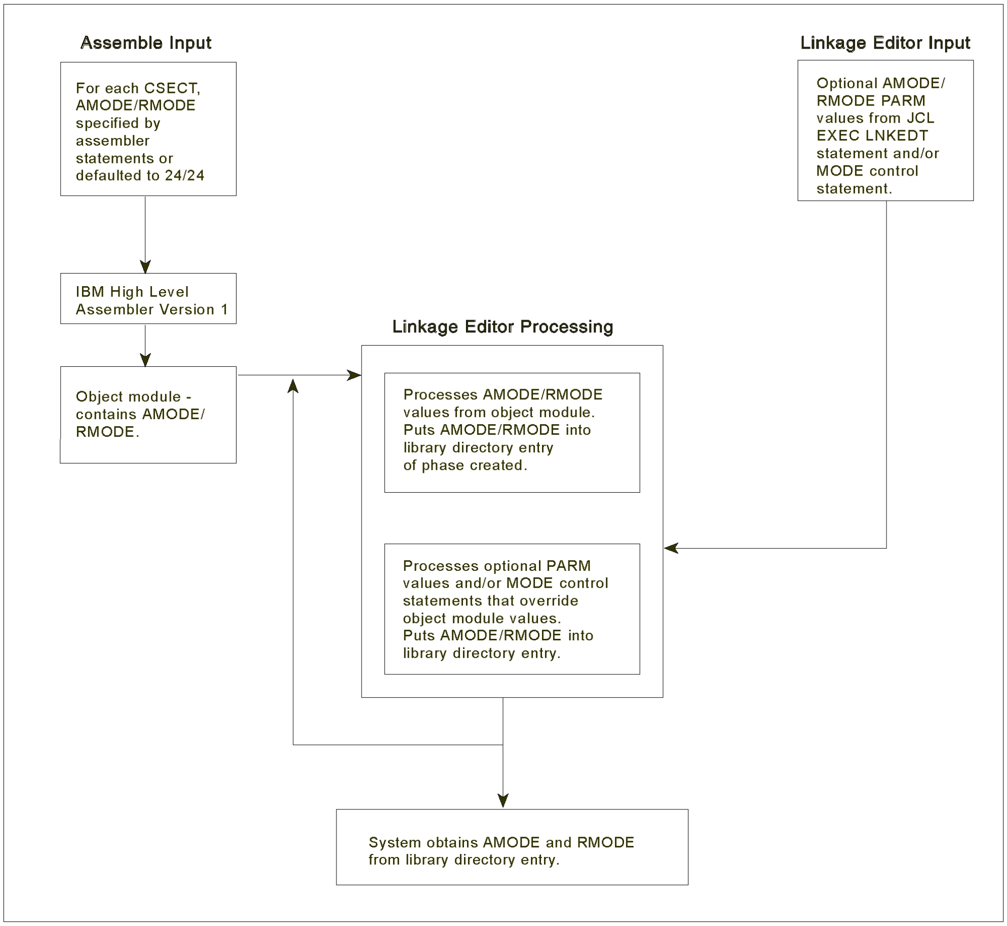

Assemble Input
For each CSECT, AMODE/RMODE specified by assembler statements or defaulted to 24/24
IBM High Level Assembler Version 1
Object module - contains AMODE/ RMODE.
Linkage Editor Input
Optional AMODE/ RMODE PARM values from JCL EXEC LNKEDT statement and/or MODE control statement.
Linkage Editor Processing
Processes AMODE/RMODE values from object module. Puts AMODE/RMODE into library directory entry of phase created.
Processes optional PARM values and/or MODE control statements that override object module values. Puts AMODE/RMODE into library directory entry.
System obtains AMODE and RMODE from library directory entry.

Figure 4. AMODE and RMODE Processing by the Linkage Editor

Note: The linkage editor uses default values of AMODE 24/RMODE 24 whenever an assembler or compiler does not support AMODE/RMODE or if they are not specified.

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

AMODE 24 RMODE 24
AMODE 31 RMODE 24
AMODE 31 RMODE ANY
AMODE ANY RMODE 24
AMODE ANY RMODE ANY

The linkage editor accepts the ANY/ANY combination from the object module but does not accept ANY/ANY from the PARM value or the MODE control statement.

The assignment of the AMODE/RMODE combination ANY/ANY to a CSECT may be useful when the CSECT is a common routine which may run in both addressing modes and may be located anywhere in storage. Refer also to the topic AMODE/RMODE Combinations from the ESD for details about assigning AMODE/RMODE to the entry point of a phase marked with AMODE ANY/RMODE ANY.

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

Linkage Editor RMODE Processing

In constructing a phase, the linkage editor is frequently requested to combine multiple CSECTs.

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

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

z/VSE treats programs in overlay structure as RMODE 24 programs. To build overlay programs, you can use PHASE statements where the origin in the statement is defined as "ROOT", as a "symbol", or as "*", but not if it is the first phase of a link-editing run. All phases of an link-editing job step containing such PHASE statements are assigned an RMODE of 24, regardless of the ESD data, the PARM field parameter, or the MODE control statement operand.

How to Change the AMODE

To change the addressing mode you must change the value of the PSW AMODE bit.

The High Level Assembler example in Figure 5 on page 19 illustrates how to make a 24-bit addressing mode program to retrieve data from an area, which might reside above 16 MB. The example works correctly whether or not the area is actually above 16 MB.

The example shows three ways of switching the AMODE:

- By using the BSM instruction,
- By using the AMODESW macro,
- By using the SAM24 and SAM31 instructions.

In the example, the L 2,4(,15) instruction must be executed in AMODE 31l, if register 15 points to an area above 16 MB. The LA 12,0(,12) instruction clears the high-order byte of the base register to ensure that its contents are correct in AMODE 31.

```
USER     CSECT
USER     RMODE 24
USER     AMODE 24
         BALR 12,0      LOAD BASE REGISTER
         USING *,12     ESTABLISH ADDRESSABILITY
         LA 12,0(,12)   CLEAR HIGH-ORDER BYTE
*
* Variant 1 - change the addressing mode using the BSM command
         L   1,LABEL1   SET HIGH-ORDER BIT OF REGISTER 1 TO 1
                        AND PUT ADDRESS INTO BITS 1-31
         BSM 0,1        SET AMODE 31 (DOES NOT PRESERVE AMODE)
LABEL1   DC  A(LABEL2 + X'80000000')
LABEL2   DS  0H
         L   2,4(,15)   OBTAIN DATA FROM ABOVE 16 MB
         LA  1,LABEL3   SET HIGH-ORDER BIT OF REGISTER 1 TO 0
                        AND PUT ADDRESS INTO BITS 1-31
         BSM 0,1        SET AMODE 24 (DOES NOT PRESERVE AMODE)
LABEL3   DS  0H
*
* Variant 2 - change the addressing mode using the AMODESW macro
         AMODESW SET,AMODE=31   SET AMODE 31 (DOES NOT PRESERVE AMODE)
         L     2,4(,15)  OBTAIN DATA FROM ABOVE 16 MB
         AMODESW SET,AMODE=24   SET AMODE 24 (DOES NOT PRESERVE AMODE)
*
*  Variant 3 - change the addressing mode using the SAM24 and SAM31
*              commands
         SAM31           SET AMODE 31 (DOES NOT PRESERVE AMODE)
         L     2,4(,15)  OBTAIN DATA FROM ABOVE 16 MB
         SAM24           SET AMODE 24 (DOES NOT PRESERVE AMODE)
```

Figure 5. Mode Switching to Retrieve Data from Above 16 MB

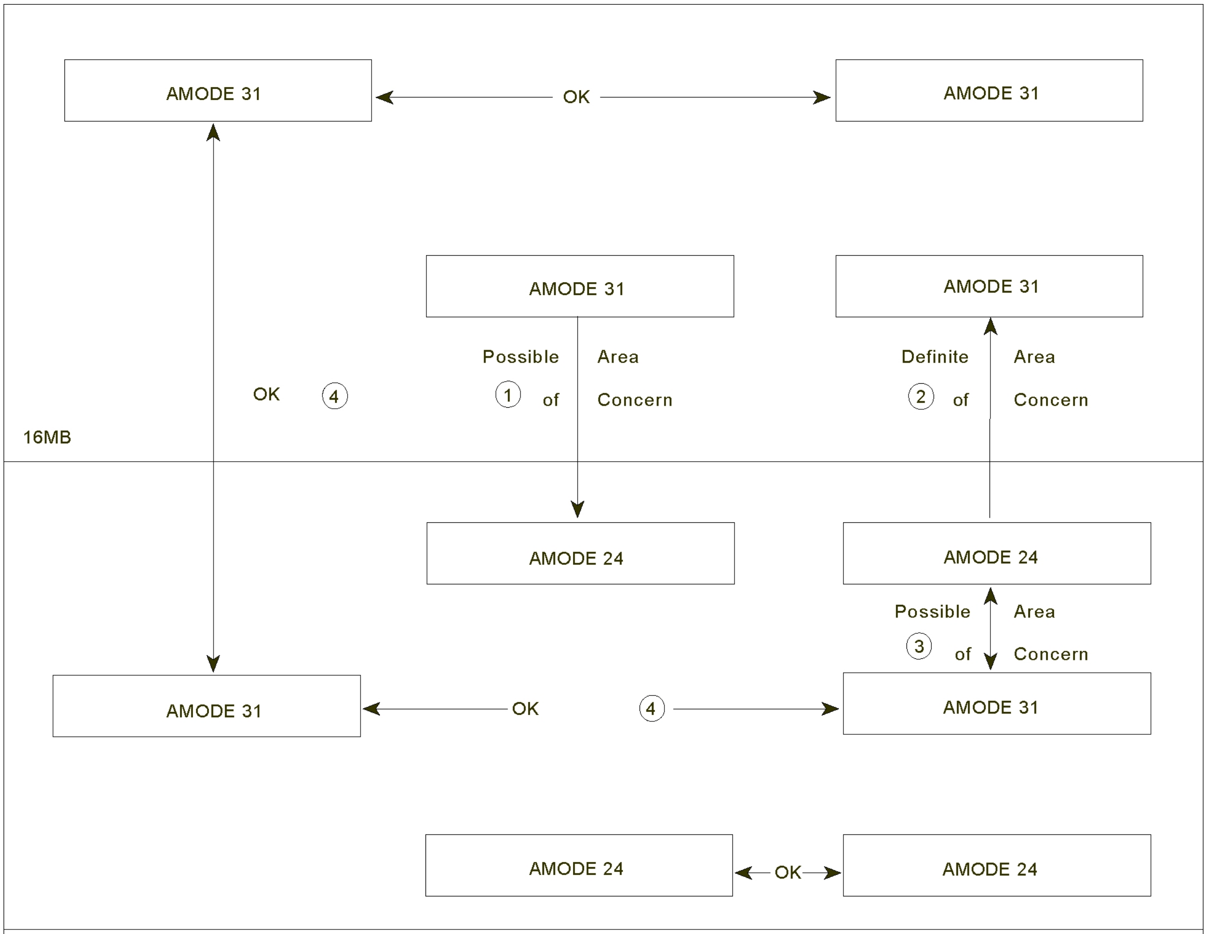
The example in Figure 5 on page 19 shows what is to be changed for switching an address mode.

Chapter 4. Establishing Linkage in a 31-Bit Addressing Environment

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

As shown in Figure 6 on page 22, it is the linkage between modules whose addressing modes are different that is an area of concern. The areas of concern that appear in Figure 6 on page 22 fall into two basic categories:

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



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

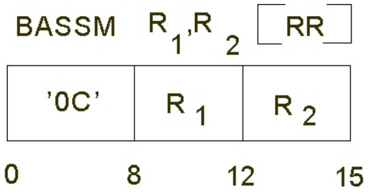
Figure 6. Linkage Between Modules with Different AMODEs and RMODEs

There are a number of ways of dealing with the areas of concern that appear in Figure 6 on page 22:

- Use the instructions BASSM and BSM
- Use the macro AMODESW
- Use pointer-defined linkage
- Use linkage assist routines
- Use “capping.”

Using the BASSM and BSM Instructions

The BASSM (branch and save and set mode) and the BSM (branch and set mode) instructions are branching instructions that set the addressing mode. They are designed to complement each other. BASSM is used to call (AMODESW CALL) and BSM is used to return (AMODESW RETURN) but they are not limited to such use. The description of BASSM appears in Figure 7 on page 23.



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

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

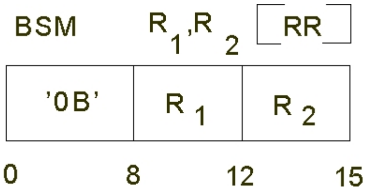
When the contents of the general register designated by the R_2 field are used, bit 0 of the register specifies the new addressing mode and replaces bit 32 of the current PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The new value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

Condition Code: The code remains unchanged.

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

Figure 7. BRANCH and SAVE and Set Mode (BASSM) Description

Refer to Figure 8 on page 24 for a description of the BSM instruction.



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

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

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

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

Condition Code: The code remains unchanged.

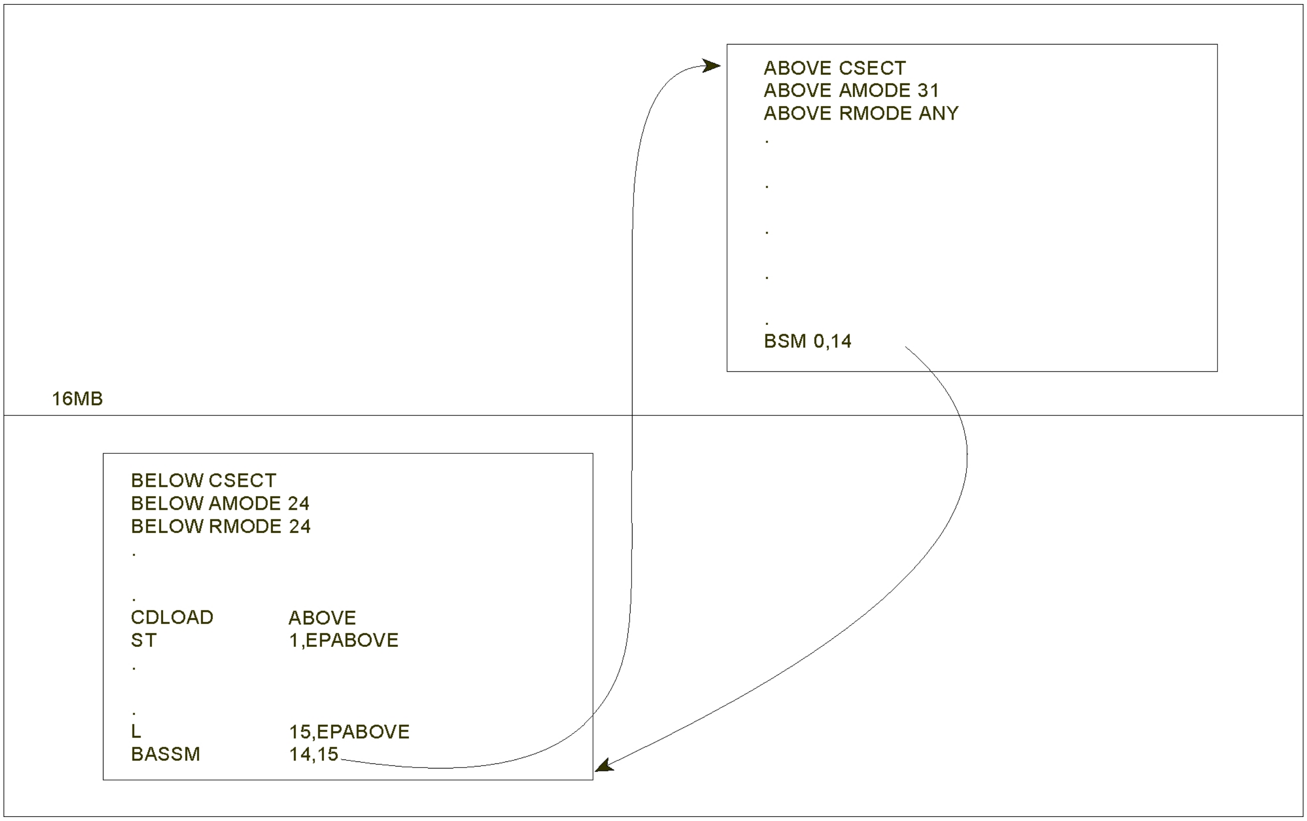
Program Exceptions: None.

Figure 8. Branch and Set Mode (BSM) Description

Calling and Returning with BASSM and BSM

In the following example, a module named BELOW has the attributes AMODE 24, RMODE 24. BELOW uses a CDLOAD macro to load phase ABOVE above 16 MB and obtain the address of phase ABOVE. The CDLOAD macro returns the address in register 1 with the AMODE in bit 0 (a pointer-defined value). BELOW stores this address in location EPABOVE. When BELOW is ready to branch to ABOVE, BELOW loads ABOVE's entry point address from EPABOVE into register 15 and branches using BASSM 14,15. BASSM places the address of the next instruction into register 14 and sets bit 0 in register 14 to 0 to correspond to BELOW's addressing mode. BASSM replaces the PSW A-mode bit with bit 0 of register 15 (a 1 in this example) and replaces the PSW instruction address with the branch address (bits 1-31 of register 15) causing the branch.

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



```
ABOVE CSECT
ABOVE AMODE 31
ABOVE RMODE ANY
.
.
.
.
.
BSM 0,14
```

```
BELOW CSECT
BELOW AMODE 24
BELOW RMODE 24
.
.
CDLOAD        ABOVE
ST            1,EPABOVE
.
.
L             15,EPABOVE
BASSM         14,15
```

Figure 9. Using BASSM and BSM

Using Pointer-Defined Linkage

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

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

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

Using an ADCON to Obtain a Pointer-Defined Value

The following method is useful when you need to construct pointer-defined values to use in pointer-defined linkages between control sections or phases that will be link edited into a single phase.

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

In Figure 10 on page 26, RTN1 obtains pointer-defined values from RTN2 and RTN3. RTN1, the invoking routine does not have to know the addressing modes of RTN2 and RTN3. Later, RTN2 or RTN3 could be changed to use different addressing modes, and at that time their address constants would be changed to correspond to their new addressing mode. RTN1, however, would not have to change the sequence of code it uses to invoke RTN2 and RTN3.

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

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

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

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

Figure 10. Example of Pointer-Defined Linkage

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

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

When an AMODE 24 program invokes a phase that is to execute in AMODE 31, the calling program must ensure that register 13 contains a valid 31-bit address of the register save area with no extraneous data in bits 1-7 of the high-order byte (3-byte address). In addition, when any program invokes an AMODE 24 program, register 13 must point to a register save area located below 16 MB.

Using the CDLOAD/LOAD Macro to Obtain a Pointer-Defined Value

CDLOAD/LOAD returns a pointer-defined value in register 1. You can preserve this pointer-defined value and use it with a BASSM instruction to pass control without having to know the target routine's AMODE.

Linkage Assist Routines

A **linkage assist routine**, sometimes called an addressing mode interface routine, is a program that performs linkage for programs executing in different AMODEs or RMODEs. Using a linkage assist routine, an AMODE 24 program can invoke an AMODE 31 program without having to make any changes. The invocation results in an entry to a linkage assist routine that resides below 16 MB and invokes the AMODE 31 program in the specified addressing mode.

Conversely, an AMODE 31 program, such as a new user program, can use a linkage assist routine to communicate with other user programs that execute in AMODE 24. The caller appears to be making a direct branch to the target program, but branches instead to a linkage assist routine that changes modes and performs the branch to the target program.

The main **advantage** of using a linkage assist routine is to insulate a program from AMODE changes that are occurring around it.

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

- Changing the programs to use pointer-defined linkage (described in "Using Pointer-Defined Linkage" on page 25).
- Adding a prologue and epilogue to a program to handle entry and exit mode switching, as described later in this topic under "Capping."

Example of Using a Linkage Assist Routine

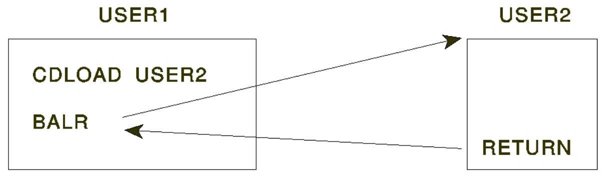
Figure 11 on page 28 shows a "before" and "after" situation involving programs USER1 and USER2. USER1 invokes USER2 by using a CDLOAD and BALR sequence. The "before" part of the figure shows USER1 and USER2 residing below 16 megabytes and lists the changes necessary if USER2 moves above 16 megabytes. USER1 does not change.

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

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

BEFORE

Existing Application - USER1 invokes USER2 repeatedly



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

AFTER

Changed Application

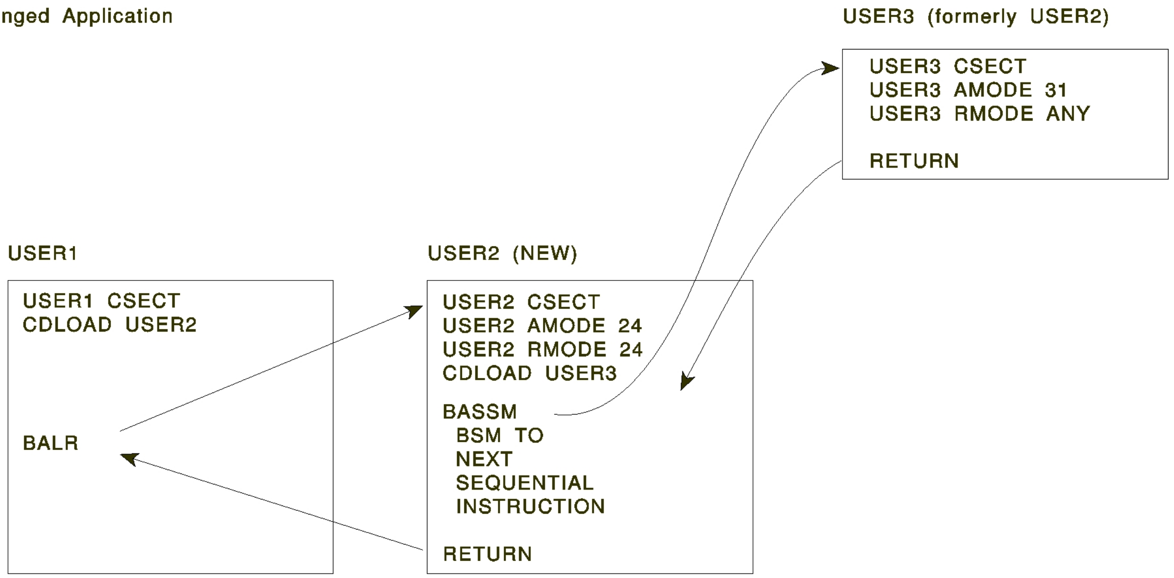


Figure 11. Example of a Linkage Assist Routine

USER1 (This module will not change)

```
*  USER MODULE USER1 CALLS MODULE USER2                          00000100
USER1    CSECT                                                   00000200
BEGIN    SAVE   (14,12)  (SAVE REGISTER CONTENT, ETC.)           00000300
*  ESTABLISH BASE REGISTER(S) AND NEW SAVE AREA (NORMAL          00000400
*  ENTRY CODING)                                                 00000500
           .
           .
*  ISSUE CDLOAD FOR MODULE USER2                                 00000700
         CDLOAD USER2    ISSUE CDLOAD FOR MODULE USER2           00000800
*  The CDLOAD macro returns a
*  pointer-defined value.  However, because module USER1
*  has not been changed and executes in AMODE 24,
*  the pointer-defined value has no effect on the BALR
*  instruction used to branch to module USER2.
         ST     1,EPUSER2   PRESERVE ENTRY POINT                 00000900
           .
*  MAIN PROCESS BEGINS                                           00001000
PROCESS  DS     0H                                               00001100
           .
           .
           .
           .
           .
           .
*  PREPARE TO GO TO MODULE USER2                                 00002000
         L      15,EPUSER2  LOAD ENTRY POINT                     00002100
         BALR   14,15                                            00002200
           .
           .
           .
           .
         TM                 TEST FOR END                         00003000
         BC     PROCESS     CONTINUE IN LOOP                     00003100
           .
         L      13,4(,13)
         RETURN (14,12),RC=0  MODULE USER1 COMPLETED             00005000
EPUSER2  DC     F'0'       ADDRESS OF ENTRY POINT TO USER2       00007000
         END    BEGIN                                            00007100
```

USER2 (Original application module)

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

USER2 (New linkage assist routine)

```
* THIS IS A NEW LINKAGE ASSIST ROUTINE                            0000100
* (IT WAS NAMED USER2 SO THAT MODULE USER1 WOULD NOT              0000200
*  HAVE TO BE CHANGED)                                            0000300
USER2    CSECT                                                    0000400
USER2    AMODE  24                                                0000500
USER2    RMODE  24                                                0000600
         SAVE   (14,12)  (SAVE REGISTER CONTENT, ETC.)            0000700
*  ESTABLISH BASE REGISTER(S) AND NEW SAVE AREA (NORMAL           0000800
*  ENTRY CODING)
*  CLEAR HIGH-ORDER BYTE OF BASE REGISTER(S) (PREPARE BASE
*  REGISTER(S) FOR 31-BIT ADDRESSING)
*  FIRST TIME LOGIC, PERFORMED ON INITIAL ENTRY ONLY,             0002000
*  (AFTER INITIAL ENTRY, BRANCH TO PROCESS (SHOWN BELOW))        0002100
           .
         CDLOAD USER3                                             0004000
*  USER2 LOADS USER3 BUT DOES NOT DELETE IT.  USER2 CANNOT
*  DELETE USER3 BECAUSE USER2 DOES NOT KNOW WHICH OF ITS USES
*  OF USER3 IS THE LAST ONE.
         ST     1,EPUSER3   PRESERVE POINTER DEFINED VALUE        0004100
           .
* PROCESS  (PREPARE FOR ENTRY TO PROCESSING MODULE)               0005000
           .
         (FOR EXAMPLE, VALIDITY CHECK REGISTER CONTENTS)
           .
           .
* PRESERVE AMODE FOR USE DURING RETURN SEQUENCE                   0007000
         LA     1,XRETURN           SET RETURN ADDRESS            0008000
         BSM    1,0                 PRESERVE CURRENT AMODE        0008100
         ST     1,XSAVE             PRESERVE ADDRESS              0008200
         L      15,EPUSER3          LOAD POINTER DEFINED VALUE    0009000
* GO TO MODULE USER3                                              0009100
         BASSM  14,15               TO PROCESSING MODULE          0009200
* RESTORE AMODE THAT WAS IN EFFECT                                0009300
         L      1,XSAVE             LOAD POINTER DEFINED VALUE    0009400
         BSM    0,1                 SET ADDRESSING MODE           0009500
XRETURN  DS     0H                                                0009600
         L      13,4(,13)
           .
           .
```

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

```
           .
           .
         RETURN (14,12),RC=0       MODULE USER2 HAS COMPLETED     0010000
EPUSER3  DC     F'0'                POINTER DEFINED VALUE         0010100
XSAVE    DC     F'0'                ORIGINAL AMODE AT ENTRY       0010200
         END                                                      0010500
```

USER3 (New Application Module)

```
*  MODULE USER3    (PERFORMS FUNCTIONS OF OLD MODULE USER2)      00000100
USER3    CSECT                                                   00000200
USER3    AMODE  31                                               00000300
USER3    RMODE  ANY                                              00000400
         SAVE   (14,12)  (SAVE REGISTER CONTENT, ETC.)           00000500
*  ESTABLISH BASE REGISTER(S) AND NEW SAVE AREA                  00000600
           .
           .
           .
           .
           .
           .
*  RESTORE REGISTERS AND RETURN                                  00008000
           .
         L      13,4(,13)
         RETURN (14,12),RC=0       MODULE USER3 HAS COMPLETED    00008100
         END                                                     00008200
```

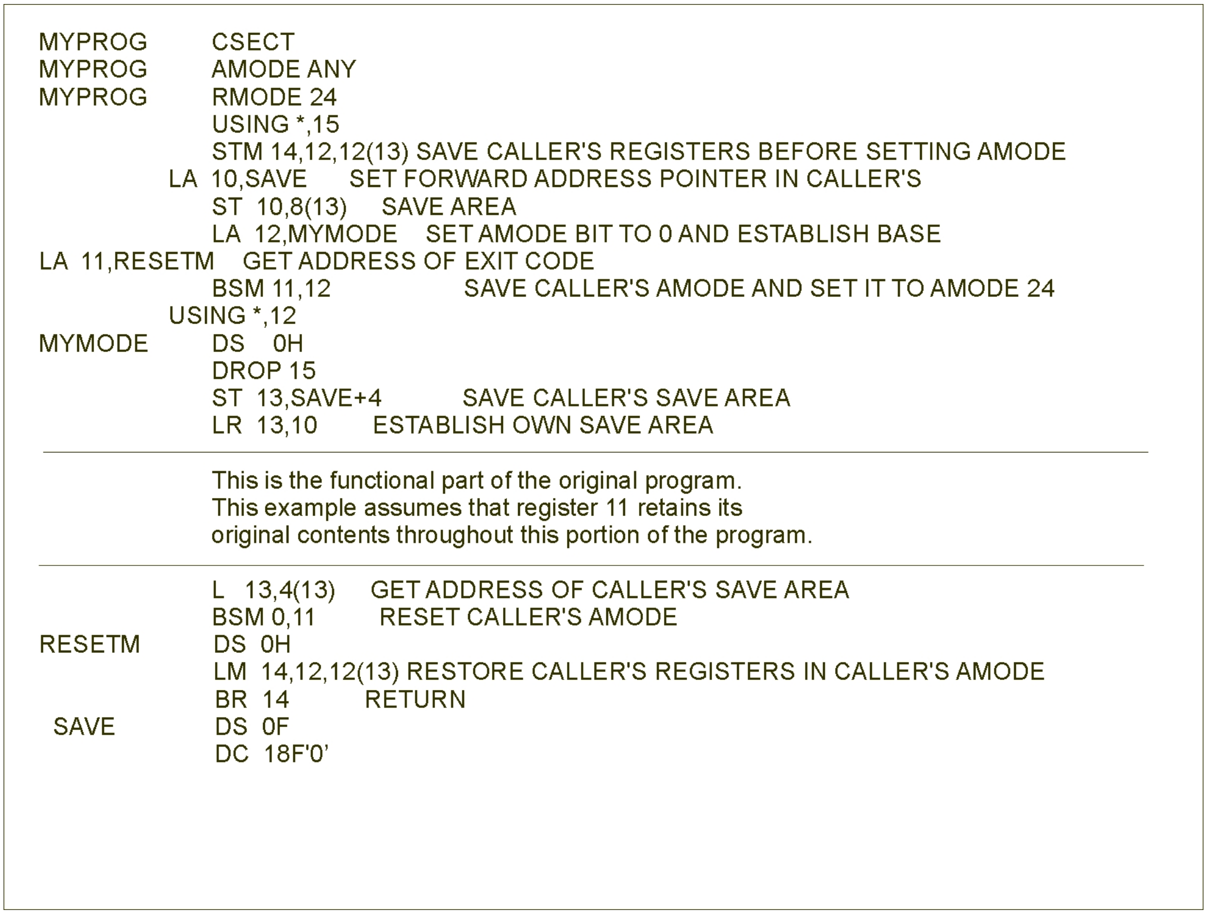
- Statements 300 and 400 establish the AMODE and RMODE values for this phase. Unless they are overridden by linkage editor PARM values or MODE control statements, these are the values that will be placed in the library directory entry for this phase.
- Statement 8100 returns to the invoking phase.

Using Capping - Linkage Using a Prologue and Epilogue

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

For example, when capping is used, a program in AMODE 24 can be invoked by programs whose AMODE is either 24 or 31; it can perform its function in AMODE 24 and can return to its caller in the caller's AMODE. Capped programs must be able to accept callers in either AMODE. Programs that reside above 16 MB cannot be invoked in AMODE 24. Capping, therefore, can be done **only for programs that reside below 16 MB**.

Figure 12 on page 31 shows a cap for an AMODE 24 program. The caller must ensure that register 15 contains a 31-bit address.



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

Figure 12. Cap for an AMODE 24 Program

Chapter 5. I/O Processing in a 31-Bit Environment

Performing I/O in 31-Bit Mode

An AMODE 31 program can perform an I/O operation by:

- Using VSE/VSAM services which accept callers in either AMODE 24 or 31.
- Using the EXCP (execute channel program) macro. All parameter lists, control blocks, CCWs, and EXCP appendage routines must reside in virtual storage below the 2 GB bar.
- Invoking a routine that executes in AMODE 24 as an interface to access methods such as SAM or DAM which only accept callers executing in AMODE 24. See Chapter 4, “Establishing Linkage in a 31-Bit Addressing Environment,” on page 21 for more information about this method.
- Using the method shown in Figure 13 on page 34.

Notes:

1. To perform I/O operations to buffers located **above** 16 MB, a program must use either:
   - The VSE/VSAM access method.
   - The EXCP macro with format 1 CCWs.
2. For all other access methods, the data buffers must be located below 16 MB.

Using the EXCP Macro for I/O to Virtual Storage Above 16 MB

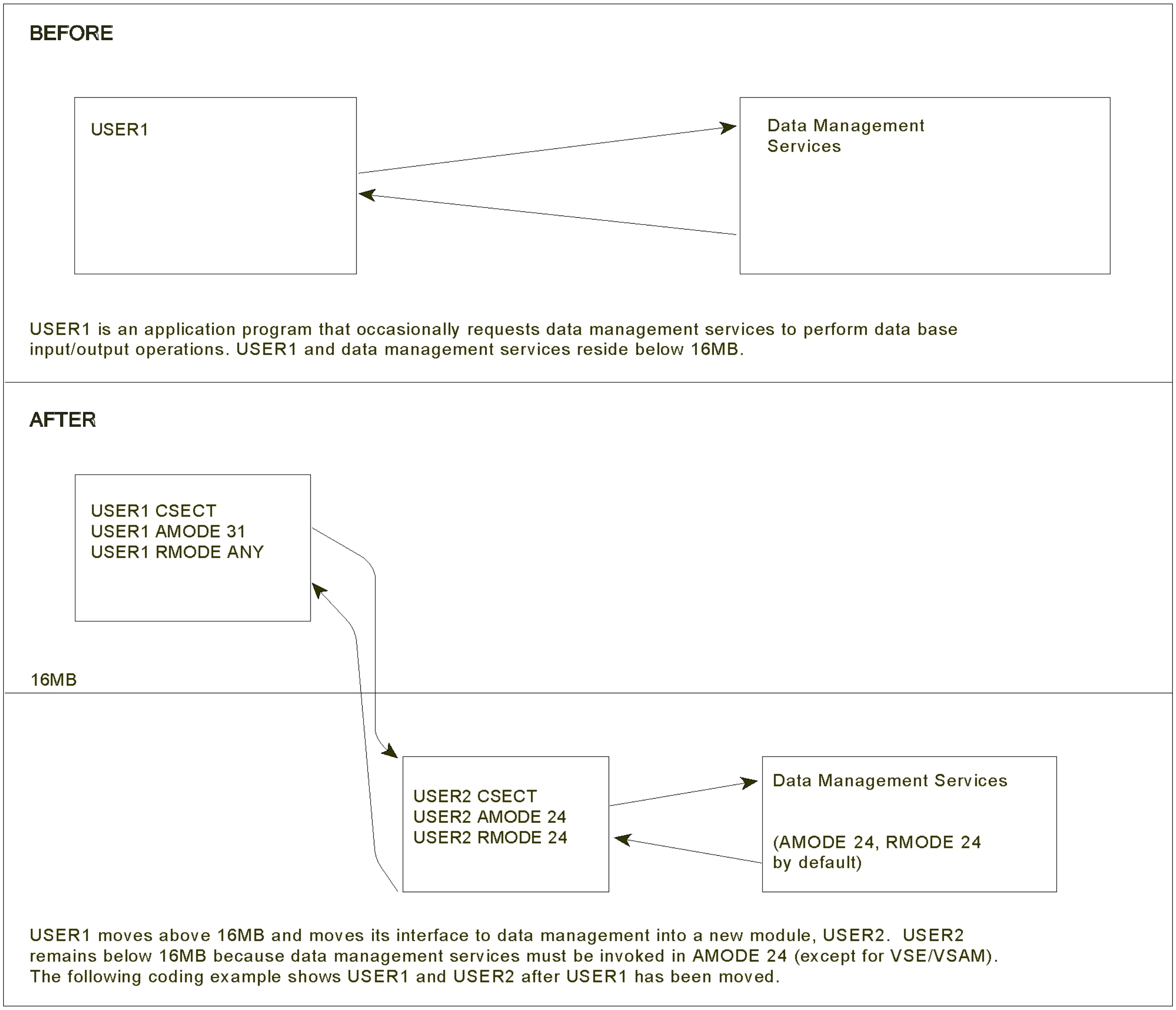
EXCP macro users can perform I/O to virtual storage areas above 16 MB. By using format 1 CCWs in the EXCP channel program, users can point to 31-bit virtual addresses of an I/O buffer.

Although the I/O buffer can be in virtual storage above 16 MB, the format 1 CCW that contains the pointer to the I/O buffer and all the other areas related to the I/O operation (such as CCBs and appendages) must reside in virtual storage below 16 MB. Refer to “I/O Processing Support for 31-Bit Addressing” on page 132 for further details about the restrictions which apply and to the manual z/VSE System Macros Reference for a detailed description of the EXCP macro.

Example of Performing I/O While Residing Above 16 MB

Figure 13 on page 34 shows a “before” and “after” situation that involves two functions, USER1 and USER2. In the BEFORE part of the example, USER1 contains both functions and resides below 16 MB. In the AFTER part of the example USER1 has moved above 16 MB. The portion of USER1 that requests data management services has been removed and remains below 16 MB.

Note: For details of how to perform I/O on memory objects residing above 2 GB, see “Using 64-Bit Virtual I/O Operations on Memory Objects” on page 53.



USER1 is an application program that occasionally requests data management services to perform data base input/output operations. USER1 and data management services reside below 16MB.

USER1 moves above 16MB and moves its interface to data management into a new module, USER2. USER2 remains below 16MB because data management services must be invoked in AMODE 24 (except for VSE/VSAM). The following coding example shows USER1 and USER2 after USER1 has been moved.

Figure 13. Performing I/O While Residing Above 16 MB

Chapter 6. Real Storage Considerations for User Programs (31-Bit Addressing)

In a system with more than 16 MB real storage, it is possible to obtain PFIXed storage above 16 MB (using the PFIX macro with RLOC=ANY or the GETVIS macro with LOC=ANY and PFIX=YES).

Programs using the REALAD macro have to be aware that this macro always returns a valid 31-bit real address.

The documentation z/VSE System Macros Reference describes the GETVIS, PFIX, and REALAD macros in detail.

Part 2. 64-Bit Addressing Support

1. The information and examples provided for 64-bit addressing in this manual are based on functions of the High Level Assembler.
2. High Level Assembler refers to *High Level Assembler Version 1.6 for z/OS, z/VM, and z/VSE*, which is a base program of Version 5 Release 1 of IBM z/VSE.

Chapter 7. Using the 64-Bit Address Space

This topic describes how you can (a) create and maintain (64-bit) *memory objects*, and (b) execute *64-bit applications*.

Note: The address space structure below the 2 GB bar is not affected by using memory objects in the 64-bit address space. All programs in AMODE 24 and AMODE 31 continue to run unchanged.

This topic contains:

Related Topics:

For details of how to:	Refer to:
• use the IARV64 macro to create and manage memory objects **• use the SDUMPX macro for problem solving when memory objects are used,** **• use the STXIT macro with parameter ANY64 to define user exits in a 64-bit environment,**	"Macro Descriptions" in z/VSE System Macros Reference.
code the assembler instructions that are used for 64-bit operations	z/Architecture Principles of Operation.

For details of how to:	Refer to:
• use the QUERY MEMOBJ command to display the: **– Details and totals of allocated private memory objects (PMOs),** **– Size of the allocated Extended Shared Area,** **– Limits and high watermarks,** **• use the MAP, MAP REAL, and MAP <SYSLOG-ID> commands to obtain information about the current use of memory objects.** **• use the STDOPT statement with option SADUMP to specify if a standalone dump should include memory objects**	"Job Control and Attention Routines" in z/VSE System Control Statements.

What is the 64-Bit Address Space?

The 64-bit address space is supported from z/VSE 5.1 onwards. Before z/VSE 5.1, z/VSE only supported the 24-bit and 31-bit address spaces:

- A 24-bit address space is an address space that is supported by 24-bit addresses. It begins at address 0 and ends at address 16 MB.
- A 31-bit address space is an address space that is supported by 31-bit addresses. It begins at address 0 and ends at address 2 GB.

A 64-bit address space is an address space that is supported by 64-bit addresses. It begins at address 0 and ends at address 16 E (16 Exabytes, which is an incomprehensibly large number). z/VSE does not support this full range of a 64-bit address space. The size of a 64-bit address space is limited by the value of *VSIZE*, which is currently 90 GB.

To maintain program compatibility, z/VSE provides *three* addressing modes (AMODEs):

- Programs that run in AMODE 24 can only use the first *16 MB* of the address space.
- Programs that run in AMODE 31 can only use the first *2 GB* of the address space.
- Programs that run in AMODE 64 can use the *complete* 64-bit address space by *explicitly* switching to AMODE 64.

In the 31-bit address space, a "virtual line" marks the 16 MB address.

The 64-bit address space includes a second "virtual line" called "the bar" that marks the *2 GB address*.

The bar separates storage below the 2 GB address, called "below the bar", from storage above the 2 GB address, called "above the bar":

- The area above the bar can only be used for *data*. Programs continue to run *below* the bar.
- There is no area above the bar that is common to all address spaces.
- IBM reserves an area of storage above the bar for special uses to be developed in the future.

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 is provided by the first 2 GB of address space:

- Before z/VSE 5.1, a program's need for additional virtual storage was sometimes met by creating one or more *data spaces*. Programs might also have used complex algorithms to manage storage, reallocate and reuse areas, and check storage availability.

- Using the 64-bit address space, these types of programming complexity are no longer required. A program can now potentially have as much virtual storage as it needs, while containing the data within the program's primary address space.

Virtual Storage Management Above the Bar

Virtual storage above the bar is organized as *memory objects* that are created by programs using the *IARV64 macro*. A memory object is a contiguous range of virtual addresses. Each memory object begins on a 1 MB boundary and is multiple of 1 MB in size.

Note: Programs continue to run and execute in the first 2 GB of the address space.

There are *two* types of memory objects:

- *Private memory objects* (PMOs) that are created within the *Extended Private Area* (EPA) and can only be accessed from the address space in which they were created. Private memory objects are described in "Creating Private Memory Objects" on page 43.
- *Shared memory objects* (SMOs) that are created within the *Extended Shared Area* (ESA) and can be accessed from any address space that requests access to the shared memory objects. Shared memory objects are describe in "Creating/Obtaining Access to Shared Memory Objects" on page 47.

Before you can use the IARV64 macro, you must use the SYSDEF MEMOBJ statement to define the *limits* for memory objects:

- MEMLIMIT defines the total amount of virtual storage that can be allocated to memory objects within the system. MEMLIMIT also limits the use of PMOs within a single address space.
- SHRLIMIT defines the total amount of virtual storage that can be allocated to SMOs within the system. It is included in MEMLIMIT.
- LFAREA defines the total amount of *real* storage that can be used to fix PMOs.
- LF64ONLY specifies that PMOs will only be fixed in the 64-bit area.

For further information about how to set MEMLIMIT, SHRLIMIT, LFAREA, and LF64ONLY, refer to z/VSE System Control Statements.

The size of the:

- EPA is equal to MEMLIMIT minus SHRLIMIT.
- ESA is equal to SHRLIMIT.

Figure 14 on page 42 shows how memory objects are used in the 64-bit address space.

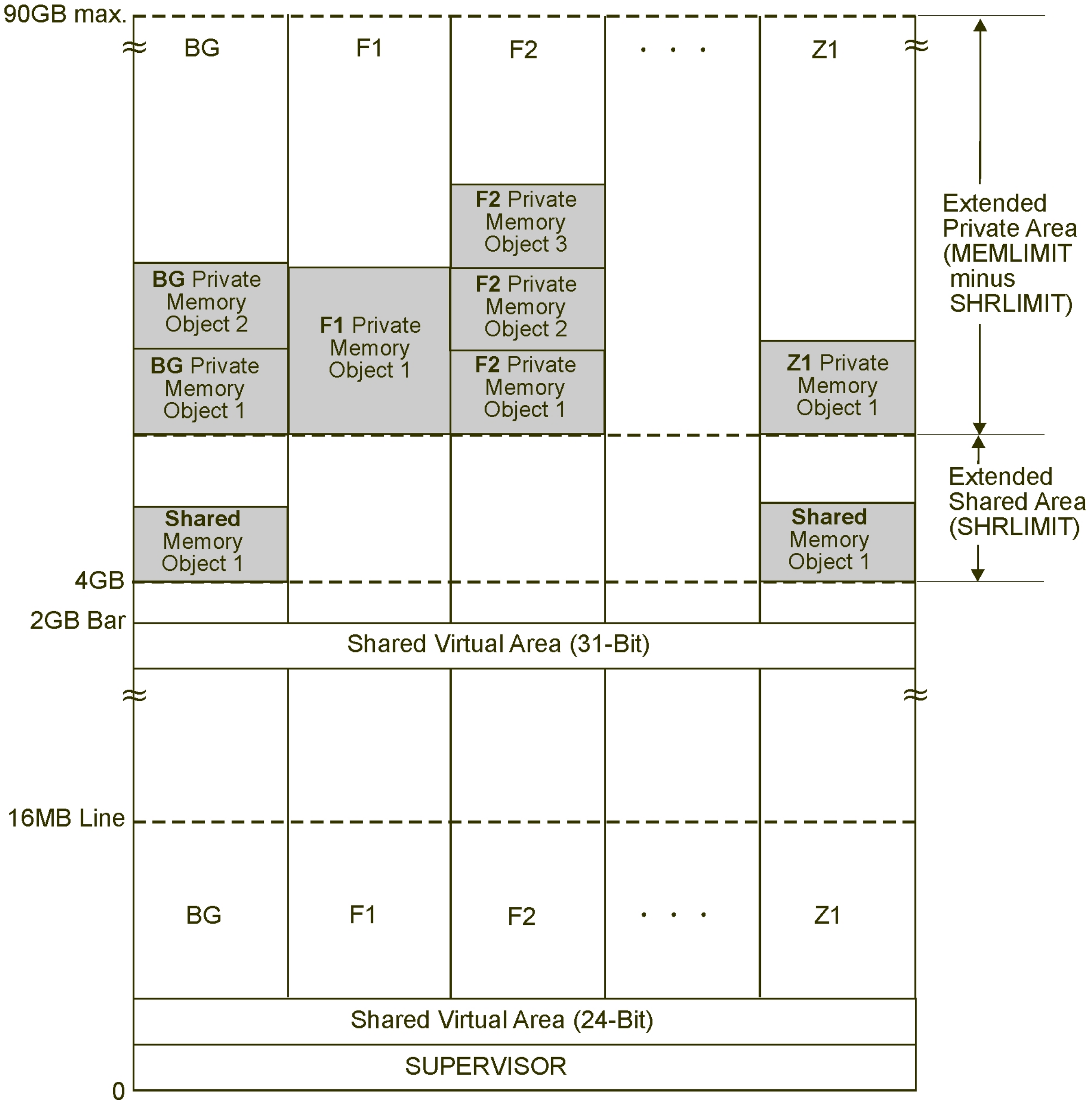

90GB max.
BG
F1
F2
. . .
Z1
F2 Private Memory Object 3
BG Private Memory Object 2
F1 Private Memory Object 1
F2 Private Memory Object 2
BG Private Memory Object 1
F2 Private Memory Object 1
Z1 Private Memory Object 1
Extended Private Area (MEMLIMIT minus SHRLIMIT)
Extended Shared Area (SHRLIMIT)
Shared Memory Object 1
Shared Memory Object 1
4GB
2GB Bar
Shared Virtual Area (31-Bit)
16MB Line
BG
F1
F2
. . .
Z1
Shared Virtual Area (24-Bit)
SUPERVISOR
0

Figure 14. Using Memory Objects in the 64-Bit Address Space

Prerequisites for Using Memory Objects

To use (64-bit) memory objects in the 64-bit address space, you require:

- z/VSE Version 5 Release 1 or later.
- Values for VSIZE, MEMLIMIT, SHRLIMIT, LFAREA, and LF64ONLY that meet your system requirements.
- User applications that have been adapted to use 64-bit memory objects.

IARV64 Macro Services and Program Rules

The IARV64 macro (which has been ported from z/OS) provides the services for using memory objects. This topic describes the use of IARV64 services in z/VSE. Programs that use the IARV64 macro with the functionality supported by z/VSE are compatible with z/OS.

Table 2 on page 43 provides an overview of the IARV64 macro services that are supported by z/VSE. There are two types of service:

- Authorized services, which require that the caller has either *supervisor state* or *PSW key* with value zero.
- Unauthorized services, that are available to a caller with *problem state*, and a *PSW key* equal to the *partition key*.

Table 2. IARV64 Services and Rules for What Programs Do with Memory Objects

IARV64 Request	A problem state, partition key program (unauthorized program)	A supervisor state or key 0 program (authorized program)
GETSTOR—create a PMO	Can create a PMO in the primary address space. The storage key of the memory object will be the same as the PSW key of the caller.	Can create a PMO in the primary address space. Can define the storage key of the PMO. Can specify whether the PMO can be freed by an authorized program only, and whether it can be PAGEFIX'd and PAGEUNFIX'd.
DETACH—free one or more PMOs	Can free a PMO it owns.	Can free a PMO it owns. Can free an SMO. Can remove a shared interest.
PAGEFIX—fix pages in one or more PMO.	Cannot fix pages.	Can fix pages in one or more PMOs.
UNPAGEFIX—undo a pagefix operation	Cannot unfix pages.	Can unfix pages in one or more PMOs.
LIST—list the memory objects.	Cannot list memory objects.	Can list memory objects in the primary address space.
GETSHARED — create an SMO.	Cannot use this service.	Can create SMOs.
SHAREMEMOBJ — requests that the specified address space be given access to one or more SMOs.	Cannot use this service.	Can use this service in the primary address space to establish addressability to the SMOs.

Note: The topics that follow describe how to use the IARV64 services. They do not describe environmental or programming requirements, register usage, or syntax rules. For this "reference" type of information, refer to the descriptions of the IARV64 macro in the z/VSE System Macros Reference.

Using Private Memory Objects

Creating Private Memory Objects

IARV64 GETSTOR is used to create a *private memory object* (PMO). PMOs are allocated in the Extended Private Area (EPA) of an address space. The size of the EPA is equal to the value of MEMLIMIT minus SHRLIMIT. The EPA only exists when there is at least one PMO allocated in the address space.

Once a program has created a PMO, all programs within the address space can access the PMO providing they have a matching key.

Note: Virtual storage within the EPA is only assigned to allocated PMOs (which is different from partition storage that has virtual storage that is permanently assigned). This means, there might be insufficient virtual storage available to meet the GETSTOR request (see description of COND parameter in “GETSTOR

Request” on page 44). This might be true even when the size of the allocated PMOs in the address space is less than the size of the EPA.

The z/VSE resources required to use the EPA of an address space (for example, control blocks) are freed when the *last PMO* within the EPA of the address space has been freed.

GETSTOR Request

To create a private memory object (PMO), use the IARV64 GETSTOR request.

Parameters for GETSTOR include:

- SEGMENTS=*segments* specifies the size, in megabytes, of the PMO you are creating. The system returns the address of the PMO in the ORIGIN parameter.
- FPROT=YES (the default) gives the PMO fetch protection.
- KEY=*key* specifies the PMO's storage key (authorized programs only).
- USERTKN=*user token* is an 8-byte token that can be associated with a PMO and relates two or more PMOs to each other. The user token can be specified on a DETACH request to free *all* PMOs that are associated with this token.

  For an unauthorized program, the high-order half (bits 0-31) of the user token must be binary zeros.

  For an authorized program, the high-order half (bits 0-31) of the user token must be non-zero.
- CONTROL=AUTH is required in order to PAGEFIX or PAGEUNFIX a PMO. When specified, a PMO can only be freed by an authorized program.
- DETACHFIXED=YES allows a PMO to be detached even if some or all pages are fixed (authorized programs only).
- COND=YES can be used to avoid a program abend occurring when insufficient resources are available to meet a GETSTOR request. For example, MEMLIMIT or virtual storage might be exceeded. When COND=YES is specified and sufficient resources are not available, the system rejects the request but the program continues to run. The IARV64 service returns to the caller with a non-zero return code.

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

Example of Creating a Private Memory Object

The following example creates a 1 MB memory object. It specifies a constant with value of one as a user token.

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

Relationship Between a Private Memory Object and Its Owner

When a program creates a private memory object (PMO), the PMO is owned by the (z/VSE) task under which the program executes.

Once a program has created a PMO, any program within the address space can access the PMO providing the program's PSW key matches the PMO's storage key.

A PMO is freed either:

- Explicitly, when the owning task frees the PMO via a DETACH request.
- Implicitly by the system, when the owning task terminates.

Fixing and Unfixing the Pages of a Private Memory Object

Authorized programs can use the IARV64 PAGEFIX request to fix (4K) pages within one or more PMOs and therefore prevent these pages from being paged out. For example, this is required when pages are referenced disabled.

Authorized programs can use the IARV64 PAGEUNFIX request to unfix pages that were previously fixed via a IARV64 PAGEFIX request. A page remains fixed until the number of PAGEUNFIX operations for that page equals the number of PAGEFIX operations.

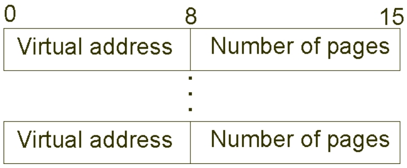
IARV64 PAGEFIX or PAGEUNFIX can only be used for PMOs that were created using the CONTROL=AUTH parameter of the GETSTOR request.

On the RANGLIST parameter, the program provides a list of virtual storage areas that are to be fixed (PAGEFIX request) or unfixed (PAGEUNFIX request). Each virtual storage area must be within one PMO.

Performance Recommendation! - You are recommended to fix contiguous areas using **one** PAGEFIX request (**one** RANGLIST entry) and *not* to specify one RANGLIST entry for each 4 KB page.

The format of the list is:

Range list format - 1 to 16 pairs



Example of Fixing Pages of a Private 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
* 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
```

Freeing a Private Memory Object

You use the IARV64 DETACH request to free one or more private memory objects (PMOs). You can only free a PMO providing the task (under which your program executes) owns the PMO.

With an IARV64 DETACH request, you can either:

- use MATCH=SINGLE,MEMOBJSTART to free a single PMO, as identified by its origin address.
- use MATCH=USERTKN,*usertkn* to free all PMOs that were created with *usertkn* on the GETSTOR request and are owned by the task under which your program executes.

Four conditions to avoid when you try to free a PMO are:

- Freeing a PMO that does not exist.
- Freeing a PMO that has PAGEFIXED pages and was created using DETACHFIXED=NO.
- Freeing a PMO that is not owned by the currently-active task.
- Freeing a PMO that has I/O in progress.

If either of the above conditions occurs and you have specified:

- COND=NO, then your program will abend.
- COND=YES, then the system will reject the DETACH request but the program continues to run. IARV64 DETACH returns to the caller with a non-zero return code. COND=YES also prevents a program abend from occurring if sufficient resources (for example, virtual storage) are not available to handle the request.

As part of normal task termination, the system frees all PMOs owned by the terminating task. If a PMO has PAGEFIXED pages, the system will (internally) unfix the pages.

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)
```

Example of Creating, Using, and Freeing a Private Memory Object

The following program creates a 1 MB private memory object (PMO) 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
```

Using Shared Memory Objects

This topic describes how you can create and use *shared memory objects* (SMOs).

SMOs are allocated in the *Extended Shared Area* (ESA). The size of the ESA is determined through SHRLIMIT. An SMO can be shared across multiple address spaces. SMO storage is similar to Shared Virtual Area (SVA) storage *except* there is no automatic addressability/access to SMO storage:

- A GETSHARED request (described in "GETSHARED Request" on page 47) creates an SMO.
- A SHAREMEMOBJ request (described in "SHAREMEMOBJ Request" on page 48) allows an address space to access an SMO.

Figure 14 on page 42 provides an overview of how SMOs are used in z/VSE.

Note: Shared memory objects can be used by *authorized* programs only.

Creating/Obtaining Access to Shared Memory Objects

This section describes how to create and obtain access to shared memory objects (SMOs).

GETSHARED Request

To *create* an SMO, use an IARV64 GETSHARED request.

Parameters for GETSHARED include:

- SEGMENTS=*segments* specifies the size, in megabytes, of the SMO you are creating. The system returns the address of the SMO in the ORIGIN parameter.
- ORIGIN= *origin* is the name or address that will contain the address of the SMO.
- FPROT=YES (the default) gives the SMO fetch protection.

- KEY=*key* specifies the SMO's storage key.
- USERTKN=*usertoken* is a required 8-byte token that is associated with the SMO. You can use a user token to relate two or more SMOs to each other. Later, you can use the user token to free all SMOs that are associated with the specified user token via one DETACH request.

  The high-order half (bits 0-31) of the user token must be non-zero.
- COND=YES can be used to avoid a program abend occurring when insufficient resources are available to meet a GETSHARED request. For example, SHRLIMIT or virtual storage might be exceeded. When COND=YES is specified and sufficient resources are not available, the system rejects the request but the program continues to run. The IARV64 service returns to the caller with a non-zero return code.

For an example of this request, see "Example of Creating and Using a Shared Memory Object – GETSHARED" on page 48.

SHAREMEMOBJ Request

To *get access to* a shared memory object (SMO), a program must use a SHAREMEMOBJ request. A SHAREMEMOBJ request creates a *shared interest* in an SMO. If an address space has a shared interest in an SMO, any program running in this address space has access to the SMO.

An address space can issue more than one SHAREMEMOBJ request for the same SMO by using different user tokens.

Parameters for SHAREMEMOBJ include:

- USERTKN=*user token* uniquely identifies the user token to be associated with the SMO (specifically, with the *shared interest* in the SMO). For a single SMO, the specified user token can be duplicated in different address spaces. However, the specified user token cannot be duplicated within a single address space for the same SMO.

  The high-order half (bits 0-31) of the user token must be non-zero.
- RANGLIST=*ranglistptr* specifies an address pointing to a range-list of SMOs that the program wants to access.

  Note: If the system detects an invalid ("non-existent") SMO in the range-list that was specified, the system will unconditionally abend the request! *None* of the specified SMOs in the range-list will be given access.
- NUMRANGE=*numrange* specifies the number of entries in the supplied range-list pointed to by RANGLIST. You can specify up to 16 SMOs.
- COND=YES can be used to avoid a program abend occurring when insufficient resources are available to meet a SHAREMEMOBJ request. When COND=YES is specified and sufficient resources are not available, the system rejects the request but the program continues to run. The IARV64 service returns to the caller with a non-zero return code.

For an example a SHAREMEMOBJ request, see "Example of Accessing a Shared Memory Object – SHAREMEMOBJ" on page 49.

For a complete listing of the IARV64 macro, refer to z/VSE System Macros Reference.

Example of Creating and Using a Shared Memory Object – GETSHARED

The following example creates a 1 MB shared memory object (SMO). 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
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 (SMO):

```
IARV64 REQUEST=SHAREMEMOBJ,
       USERTKN=USERTKNS,
       RANGLIST=RLISTPTR,
       NUMRANGE=1,
       COND=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
```

Relationship Between a Shared Memory Object and Its Owner

When your program creates a shared memory object (SMO), you need to understand the ownership of the SMO to prevent illegal operations:

- A program creates an SMO, but it does not own the SMO. An SMO is always owned by the system. This is referred to as *system affinity* (or *system interest*). System affinity for an SMO must be explicitly removed via a DETACH␠AFFINITY=SYSTEM request, which can be done by *any* authorized program in the system.
- A program gains access to an SMO by creating a *shared interest* in the SMO via a SHAREMEMOBJ request. A shared interest is owned by the main task within an address space. Shared interest for an SMO can be removed via a DETACH␠AFFINITY=LOCAL request. This can be done by *any* authorized program in the address space.
- When an address space has a shared interest in an SMO, any program running in this address space has access to the SMO providing it has a matching key.
- An address space can issue more than one SHAREMEMOBJ request for the same SMO by using different user tokens.

When the main task terminates, the system removes all shared interests owned by the main task. This means:

- Programs in this address space can no longer access SMOs.
- If this address space was the only address space with a shared interest in an SMO, and the system affinity has been removed from the SMO, the system will free the SMO.
- The memory object is no longer available for use.

Freeing a Shared Memory Object

To free a shared memory object (SMO), two actions are required:

- All address spaces must remove shared interests from the SMO by issuing a DETACH AFFINITY=LOCAL request. Shared interests from an address space are implicitly removed when the main task terminates.
- The system interest must be removed from the SMO by issuing a DETACH AFFINITY=SYSTEM request.

A DETACH request can either be used with:

- MATCH=SINGLE, in which case a single SMO will be detached.
- MATCH=USERTOKEN, in which case all SMOs associated with the specified user token will be detached.

The above applies to both AFFINITY=LOCAL and AFFINITY=SYSTEM requests.

AFFINITY=LOCAL

Using the AFFINITY=LOCAL parameter, the system removes the *shared interest* from a specified SMO within the address space. Next, one of the following can happen:

- If this address space has no further shared interests in the SMO, the address space will no longer have access to the SMO.
- If no other shared interests in the SMO exist (in either the current address space or other address spaces) and a DETACH AFFINITY=SYSTEM has been done for the SMO, the SMO is freed and is no longer available for use.
- If other shared interests in the SMO remain or DETACH AFFINITY=SYSTEM has not been done for the SMO, the SMO is not freed.

The following example removes the shared interest of the address space in the specified SMO:

```
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
```

AFFINITY=SYSTEM

Using the AFFINITY=SYSTEM parameter, the *system affinity* (or *system interest*) for the specified SMO is removed. The SMO will be freed when there is no remaining shared interest in the SMO.

Note: After the system affinity has been removed from an SMO, any further SHAREMEMOBJ requests will abend your program!

The following example frees the system affinity (system interest) in the SMO:

```
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 SMOs. Otherwise, the system might abend your program.

Here is an example of strict serialization not being maintained:

- Tasks A obtains an SMO via a GETSHARED request.
- Tasks B and C share the SMO via SHAREMEMOBJ requests.
- Task C removes the shared interest via a DETACH AFFINITY=LOCAL request.
- Task A removes system affinity via a DETACH AFFINITY=SYSTEM request. Since task B still holds a shared interest in the SMO, the SMO is not freed.
- If task C tries to share the SMO again (via a SHAREMEMOBJ request), the system will issue an abend code DC2 with reason code xx0040xx. This is because task C was not serialized against the DETACH AFFINITY=SYSTEM request issued by task A.

User Tokens and Detach Processing

A program can use the same user token for both private memory objects (PMOs) and shared memory objects (SMOs). When a program issues the following request:

```
IARV64 DETACH AFFINITY=LOCAL MATCH=USERTOKEN
```

the processing depends on the program's authorization:

- For an *unauthorized* program, all PMOs associated with the specified user token will be freed.
- For an *authorized* program, all PMOs will be freed, and all shared interests associated with the specified user token will be removed.

Protecting Storage Above the Bar

To limit access to a memory object, the creating program can use the FPROT and KEY parameters on IARV64.

- KEY assigns the storage key for the memory object.
- 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.

Dumping Memory Objects

SADUMP and SDUMPX can be used to dump memory objects:

- A specific SADUMP option can be used to include memory objects in a stand-alone dump. For details, refer to z/VSE System Control Statements.
- SDUMPX can be used to provide a dump of memory objects. For details, refer to z/VSE System Macros Reference.

In addition, refer to z/VSE Diagnosis Tools for details of how to dump memory objects using `OPTION MODUMP` or `STDOPT MODUMP=YES`.

Using the Storage in 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 56 for ways to get into AMODE 64.

The IARV64 macro is the only macro that can be called in AMODE 64. All other macros can *only* be called 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 56.

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

Listing Information About 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 IARV64WA.

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.

To request a list of SMOs defined for the system via a GETSHARED request, specify V64SHARED=*YES*.

Using a 64-Bit Application in z/VSE

A 64-bit application is program that (partly or completely) executes in AMODE 64:

- The AMODE 64 attribute is not supported for these compilers:
  - HLASM
  - COBOL
  - PL/1
  - C
  - RPG
- The AMODE 64 attribute is not supported for the linkage editor.
- The IARV64 macro is the only macro that can be called in AMODE 64.
- All other z/VSE system services (Supervisor, VSAM, BAM, DL/I, and so on) must be called in AMODE 24 or AMODE 31.
- Space-switching program calls (ss-PCs) are not supported in AMODE 64.
- Data areas for system services must be allocated below the bar.
- 64-bit addressing is not supported in VSE/ICCF pseudo-partitions.
- z/VSE provides limited support only for CICS® partitions. CICS services do *not* support 64-bit registers. Therefore, IBM recommends that you do not use 64-bit registers in CICS transactions. However, if you do wish to use 64-bit registers in a CICS transaction:
  - The CICS transaction must initialize the 64-bit registers before they are used.
  - The CICS transaction should not issue CICS services, *except* that the high-order half of the 64-bit registers must be saved before the call of the CICS service, and then restored after returning from the CICS service.
  - The CICS transaction may clear 64-bit registers if they are no longer required.
  - 64-bit registers may be used by sub-tasks that are attached in the CICS partition.

Using 64-Bit Applications and 64-Bit Operations

This topic describes the considerations for register-saving when using 64-bit applications and 64-bit operations.

- *Register saving*:
  - If a user program is interrupted, z/VSE will store the extended 64-bit registers.
  - The low-order half of the registers will be stored in the *problem program save area* that is located at start of the z/VSE partition.
  - The high-order half of the registers will be stored (in the sequence R0 to RF) in an *extended task save area*.
  - The pointer to the *extended task save area* can be obtained via a GETFLD service. For details, refer to the manual *z/VSE Supervisor Calls and Internal Macros* which you can obtain at the (whose URL is given in "Where to Find More Information" on page xiii).
- *Exit routines*:

- z/VSE exit routines and exit services belonging to the *z/OS Family API* provide 64-bit register support.
- z/VSE does not store the high-order half of the registers for Vendor exits.

Using 64-Bit Virtual I/O Operations on Memory Objects

Programs can use the EXCP macro to perform *I/O operations* to/from virtual storage above the 2 GB bar. To do so, your program must:

1. Create a private memory object (PMO) using an IARV64 REQUEST=GETSTOR call (described in "Creating Private Memory Objects" on page 43). The storage key of the PMO must be equal to the partition key.
2. Prepare CCWs (channel command words) that provide a 64-bit virtual address by specifying (a) the IDA (Indirect Data Addressing)-bit and (b) a data address that points to a single virtual Format-2 IDAW.
3. Create a CCB indicating Format-2 IDAW by using the CCB macro with the parameter IDAW=Format2.
4. Issue an EXCP request using the EXCP macro in AMODE 24 or AMODE 31. All parameter lists, control blocks, IDAWs, CCWs, and EXCP appendage routines must remain 31-bit addressable.

When the I/O operation is complete, your program can then detach the memory object using an IARV64 REQUEST=DETATCH call (described in "Freeing a Private Memory Object" on page 46).

Restrictions When Performing Virtual I/O Operations on Memory Objects:

- Only the EXCP macro can be used, which must be executed below the 2 GB bar.
- I/O operations can only be done on *private memory objects* (PMOs). I/O operations on shared memory objects (SMOs) are *not* supported.
- DASD (ECKD) devices only are supported.
- FBA-SCSI devices are *not* supported.
- Tape devices are *not* supported.
- LIOCS (logical input output control system) is *not* supported.

For further details, see "Macro Support for 64-bit Addressing" on page 134.

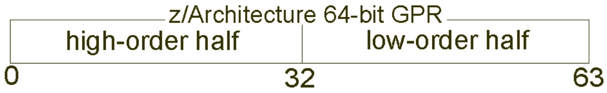
Using Assembler 64-bit Binary Operations

If you want to enhance old programs or design new ones to use the virtual storage above the 2 GB bar, you will need to use 64-bit binary operations in 64-bit address spaces

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 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.

z/Architecture 64-bit GPR
0
63

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**.



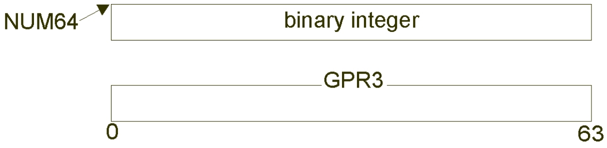
The topic provides an overview of how you can use the 64-bit GPR and the 64-bit instructions to save registers, perform arithmetic operations, access data. It explains some of the concepts that provide the foundation you need. However, for detailed information you should refer to the z/Architecture Principles of Operation:

1. Read the introduction to z/Architecture that appears in the first topic of that book.

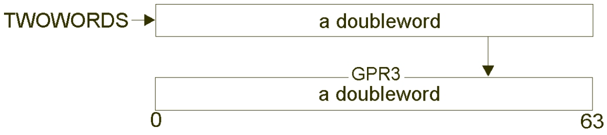
2. Then refer to the specific instructions you need to write your program.

z/Architecture Instructions That Use the 64-Bit GPR

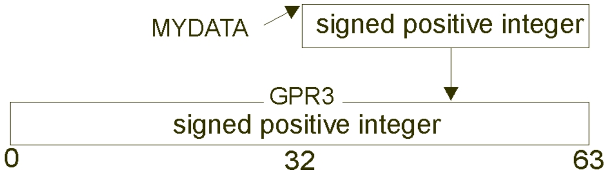
z/Architecture provides many new instructions that use two 64-bit binary integers to produce a 64-bit binary integer. These instructions include a "G" in the instruction mnemonic (AG and LG). Consider the example of an Add G instruction: AG R3,NUM64. This instruction takes the value of a doubleword binary 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.



z/Architecture 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.

Using the Assembler 64-bit Addressing Mode

If you want to enhance existing programs or design new ones to use the virtual storage above the 2 GB bar, you will need to use the 64-bit addressing mode in 64-bit address spaces.

When generating addresses, the processor performs the following address arithmetic:

1. The processor adds these three components:
   a. The contents of the 64-bit GPR.
   b. The displacement (a 12-bit value).
   c. (Optionally) the contents of the 64-bit index register.
2. 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.

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 address space.
- A program running in AMODE 24 can use only storage operands that reside in the first 16 MB of the address space.

Non-Modal Instructions

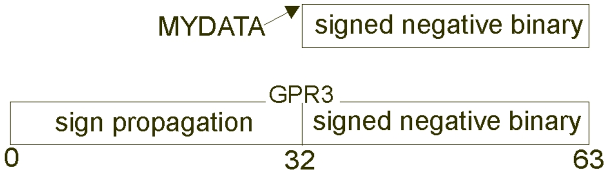
An instruction that behaves the same, regardless of the AMODE of the program, is called a *non-modal* instruction. For a non-modal instruction, AMODE can only determine where the storage operand is located.

Two good 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:

1. Adds the value of a doubleword binary integer at location NUM64 to the contents of GPR3.
2. Places 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:



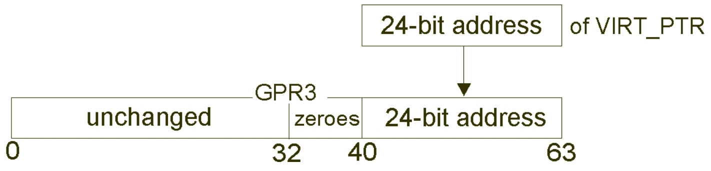
If the current AMODE is 64, MYDATA can reside anywhere in the address space; if the AMODE is 31, MYDATA must reside below 2 gigabytes; if the AMODE is 24, MYDATA must reside below 16 MB.

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.

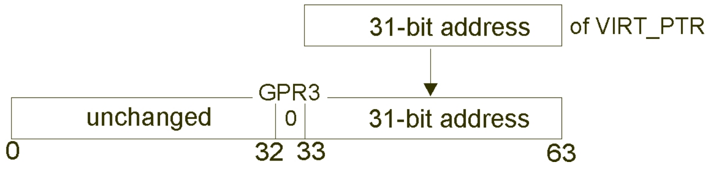
Modal Instructions

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?

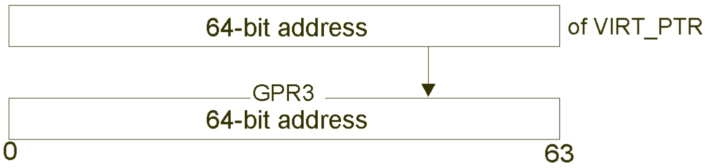
- *In AMODE 24*, the address of VIRT_PTR is a 24-bit address that is loaded into bits 40 through 63 of GPR3 (or bits 8 through 31 of the 32-bit register imbedded in the 64-bit GPR). The processor places zeros into bits 32 through 39, and leaves the first 31 bits unchanged, as follows:



- *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* Set Addressing Mode (SAM) instructions that allow you to change the addressing mode:

- SAM24, which changes the current AMODE to 24.
- SAM31, which changes the current AMODE to 31.
- SAM64, which changes the current AMODE to 64.

The AMODE bits in the PSW inform the processor as to which AMODE is currently being used.

- You can obtain the current addressing mode of a program by using the Test Addressing Mode (TAM) instruction.
- In reply, the TAM sets a condition code based upon the setting in the PSW:
  - 0 indicates AMODE 24.
  - 1 indicates AMODE 31.
  - 3 indicates AMODE 64.

SAM64, BASSM, and BSM are the *only* ways you can set the AMODE to 64. z/VSE does *not* support:

- The AMODE 64 assembler instruction.
- The linkage editor AMODE 64 statement.
- The setting up of a target routine to be given control in AMODE 64.

Linkage Conventions

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

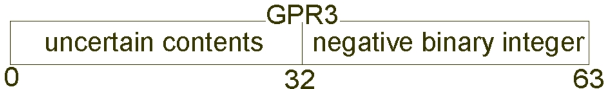
An older program changing from AMODE 31 to AMODE 64 to exploit z/Architecture instructions should expect to receive 31-bit addresses and the 72-byte save area from its callers. If you are running in AMODE 64 and want to use an address a caller has passed to you, the high-order half of the GPR will probably 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):

1. Adds the doubleword at the location MYDOUBLEWORD to the contents of the GPR3.
2. 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.

Part 3. Data Spaces and Virtual Disks

1. Before using these functions you should be familiar with the planning information provided in the topic "Using Data Spaces and Virtual Disks" in the z/VSE Planning.
2. The information and examples provided for data spaces in this manual are based on the use of the High Level Assembler.
3. High Level Assembler refers to *High Level Assembler Version 1.6 for z/OS, z/VM, and z/VSE*, which is a base program of Version 5 Release 1 of IBM z/VSE.

Chapter 8. Introducing Data Spaces

A **data space** is a range of up to 2GB of contiguous virtual storage addresses that a program can directly manipulate through z/Architecture instructions. Unlike an address space, a data space contains **only** data; it does not contain shared 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.

Terminology Use: This publication uses the term "**extended addressability**" in a general way to summarize the capabilities of 31-bit addressing, data spaces, and virtual disks. In the following discussion, however, the term "extended addressability" means basically the use of data spaces and implies in its wider context also address spaces and certain functions they support such as the use of access registers.

Whether your application is one that can use *extended addressability* depends on many factors. One basic factor is the amount of processor (real) storage available at your installation to back up virtual storage. Extended addressability frequently requires additional amounts of virtual storage, which means that your processor must have sufficient real storage available.

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

- Performance.
- Efficient use of system resources, such as storage, and the use of disk devices.
- Ability to randomly access very large amounts of data.
- Data integrity and isolation.

  Data in an address space is generally available to all tasks running in that address space; access to data in a data space can be restricted. Code running in an address space can accidentally overlay data; because of its isolation, data in a data space 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 disk device 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, 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 disk device storage for intermediate work file 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 work file on a disk device. This new solution could both improve performance and reduce the effort required for program development.

This admittedly trivial 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.

The ASC Modes

The ASC (**address space control**) 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. However, for programs accessing data in storage it is the **access register (AR) ASC mode** that is most important. In this mode, an access register identifies the address or data space the processor is to use to resolve an address. In AR ASC mode, a program can use the full set of z/Architecture instructions to manipulate data in another address space or in a data space. 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.

Terminology Use: In the following topics the short forms **primary mode** and either **access register mode** or **AR mode** are used.

In AR 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.

AR Mode and Data Spaces

Programs that access data in data spaces must run in AR mode. They use macros to create, control, and delete data spaces. z/Architecture instructions of a program executing in an address space can directly manipulate data that resides in a data space.

An Example of Using a Data Space

Suppose an existing program updates several rate tables that reside on disk. 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 disk. 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 z/Architecture 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.

Chapter 9. Using Access Registers

You cannot use access registers to branch into another address space. Through access registers, however, you can use the z/Architecture instruction set to manipulate data in other address spaces and in data spaces.

In addition to this topic, two sources of information can help you understand how to use access registers:

- Chapter 10, "Creating and Using Data Spaces," on page 81 contains examples of using access registers to manipulate data in data spaces.
- The *z/Architecture Principles of Operation* manual contain descriptions of how to use the instructions that manipulate the contents of access registers. Refer to "Where to Find More Information" on page xiii for the complete book titles.

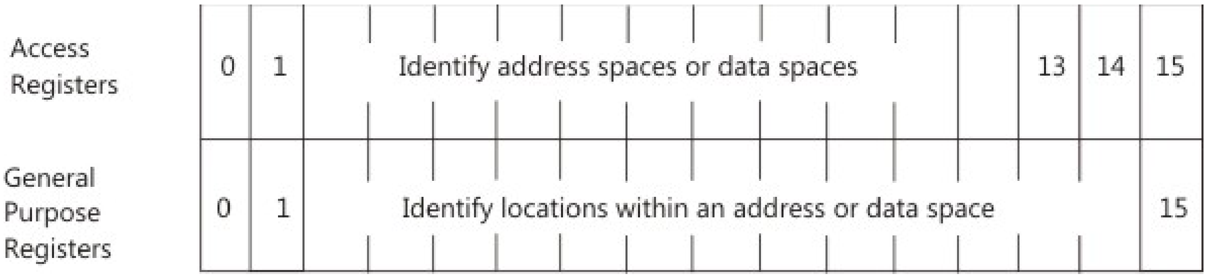
Using Access Registers for Data Reference

Through access registers, your program, whether it is in supervisor state or problem state, can use z/Architecture 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

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).


Access Registers
0
1
Identify address spaces or data spaces
13
14
15
General Purpose Registers
0
1
Identify locations within an address or data space
15

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 one 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 mode and AR mode. The PSW (program status word) 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. 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 **access list entry token (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 book 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 book, the AR and GPR pair that is used to resolve an address is called **AR/GPR**. Figure 15 on page 64 illustrates AR/GPR 4.

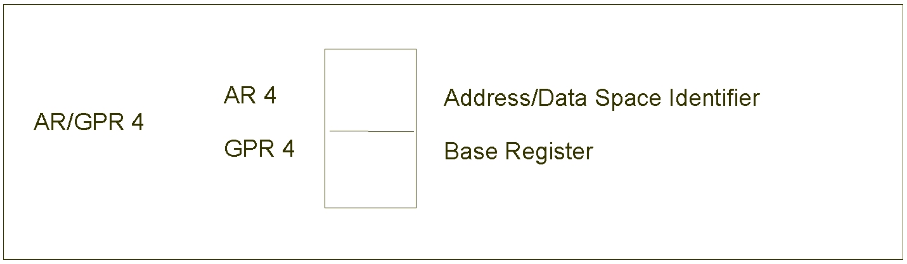


Figure 15. Example of an AR/GPR

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 MB in that space.

How does your Program Switch ASC Mode?

Use the SAC instruction to change the 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 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.

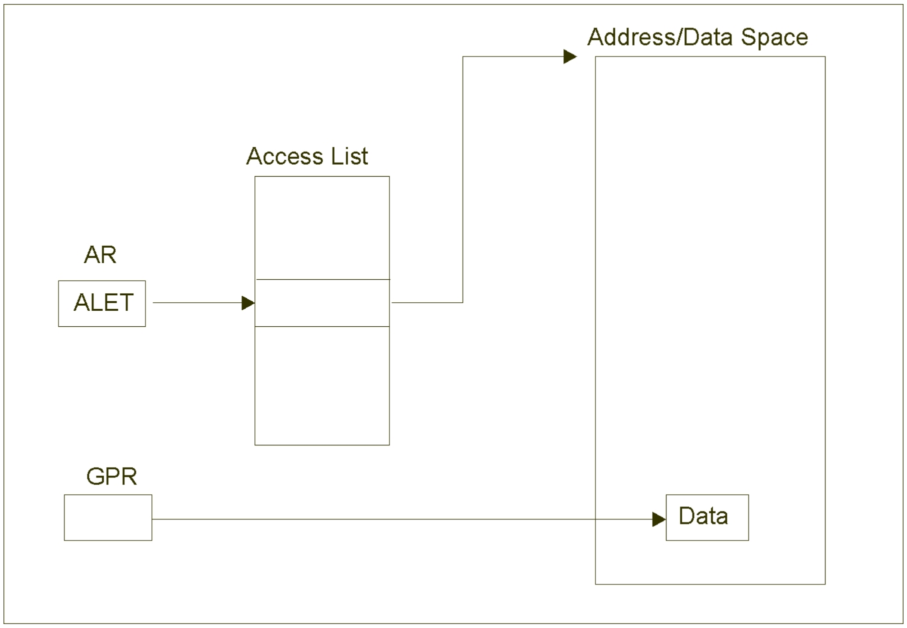


Figure 16. Using an ALET to Identify an Address/Data Space

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 17 on page 66 and Figure 18 on page 67 show two ways an MVC instruction works to move data at location B to location A.

In Figure 17 on page 66, 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.

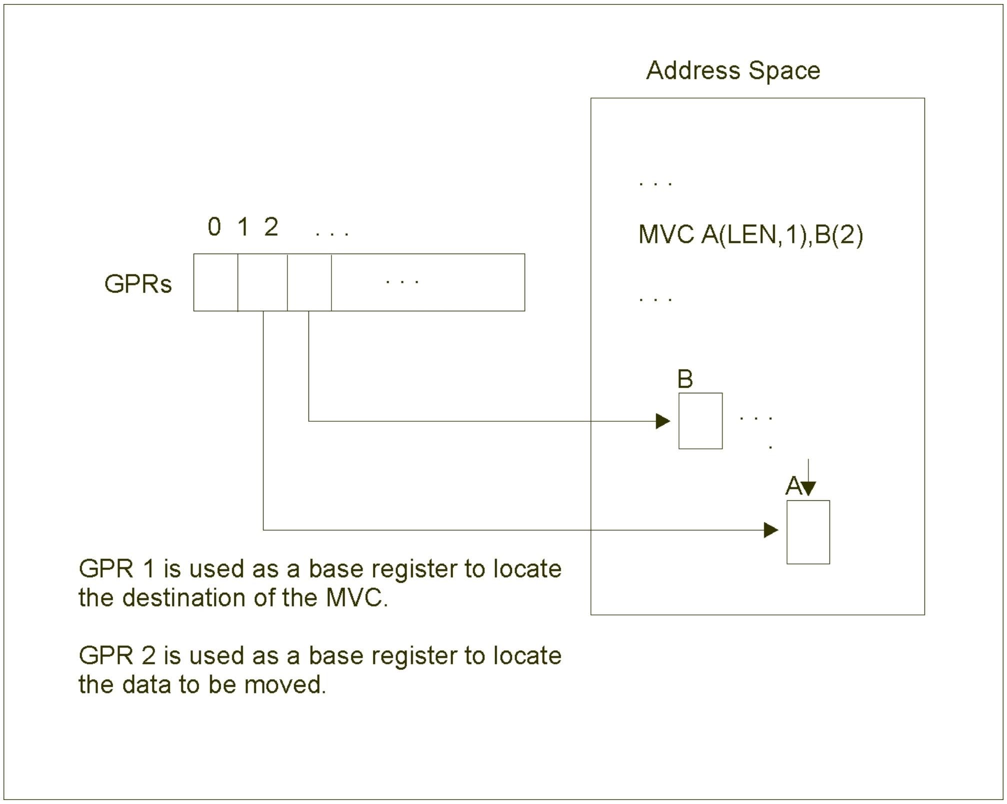

Address Space
. . .
MVC A(LEN,1),B(2)
. . .
0 1 2 . . .
GPRs
. . .
B
. . .
A
GPR 1 is used as a base register to locate the destination of the MVC.
GPR 2 is used as a base register to locate the data to be moved.

Figure 17. The MVC Instruction in Primary Mode

In Figure 18 on page 67, the MVC instruction, in code that is in AR mode, moves the data at location B in Space X to location A in Space Y. GPR 1 is used as a base register to locate the data to be moved, and AR 1 is used to identify Space X. GPR 2 is used to locate the destination 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.

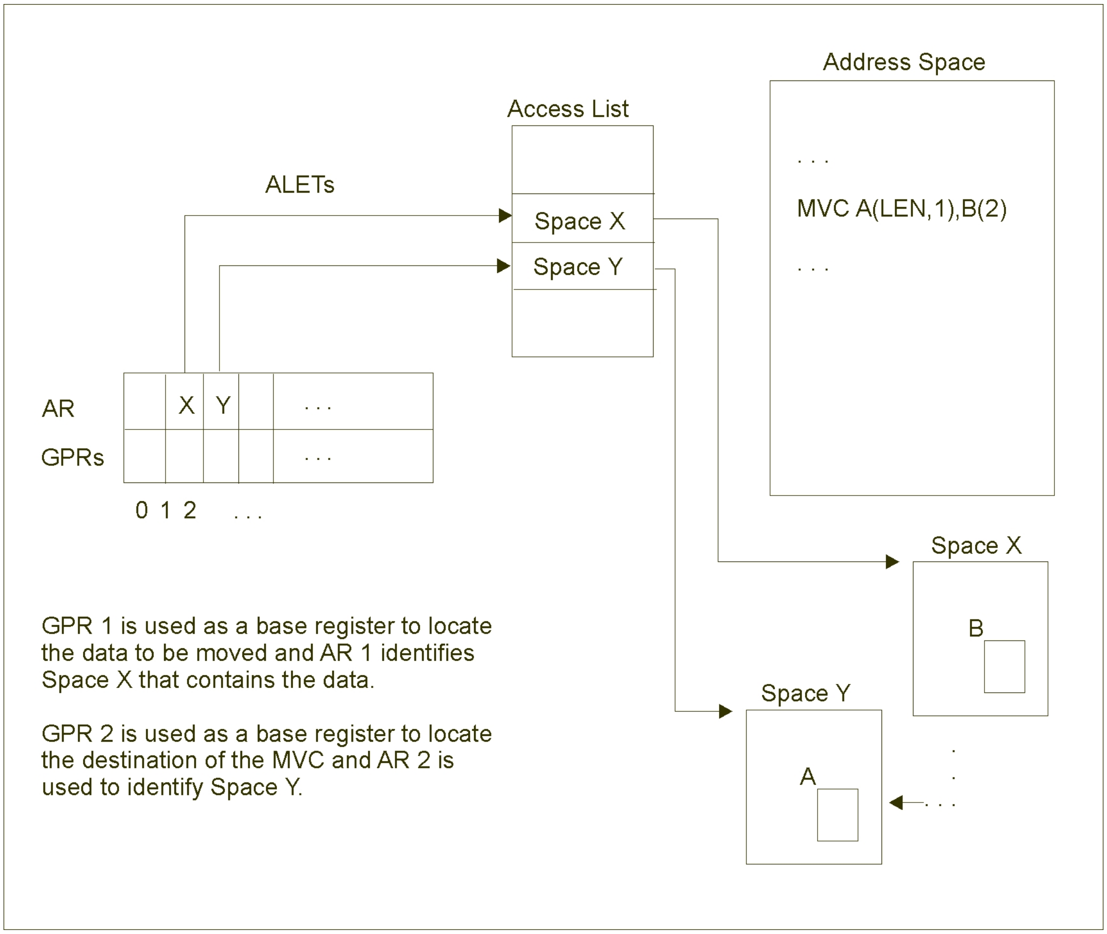

Address Space
Access List
ALETs
Space X
Space Y
. . .
MVC A(LEN,1),B(2)
. . .
AR
X
Y
. . .
GPRs
. . .
0 1 2 . . .
Space X
B
GPR 1 is used as a base register to locate the data to be moved and AR 1 identifies Space X that contains the data.
GPR 2 is used as a base register to locate the destination of the MVC and AR 2 is used to identify Space Y.
Space Y
A
. . .

Figure 18. 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 and warnings 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.

Since ARs that are associated with index registers are ignored when you code z/Architecture 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 3 on page 67 shows four examples of how a misplaced comma can change how the assembler resolves addresses on the load instruction.

Table 3. 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,)	GPR 3 is the index register. Because there is no base register, data is fetched from the primary address space.

Table 3. Base and Index Register Addressing in AR Mode (continued)

Instruction	Address Resolution
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 3 on page 67:

- 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 3 on page 67:

- 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.

Using z/Architecture Instructions to Manipulate the Contents of Access Registers

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, refer to *z/Architecture Principles of Operation* manual.

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 dispatches a z/VSE task (main task or subtask), for the first time, it assigns to that work unit a **DU-AL** (dispatchable unit access list). When the system allocates a partition, that partition gets a **PASN-AL** (primary address space number access list) assigned.

Note: Although it is called *primary address space number access list*, in z/VSE the PASN-AL is associated with a partition (residing in a particular address space).

Programs add entries to the DU-AL and the PASN-AL. The entries represent the address or data spaces that the programs want to access.

Before your program can use ARs to reference data in an address or data space, it must establish a connection to the address or data space.

Note: Only z/VSE subsystems such as CICS/VSE*, VTAM* and VSE/POWER can establish connections to address spaces.

The connection between the program that the z/VSE task represents and the address spaces and data spaces is through an access list. The process of establishing this connection is called establishing addressability.

Establishing addressability to an address or data space means your program must:

- Have authority to access data in the address or data space
- Have an access list entry that points to the address or 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.

Note: The ALESERV macro is available only to control data space entries in the access list.

The term **STOKEN** (for "space token") identifies a data space. It is similar to a partition identifier with two important differences: the system does not reuse the STOKEN value within an IPL, and data spaces do not have space IDs. The STOKEN is an eight-byte variable that the system generates when you create a data space (note that the system never generates a STOKEN value of zero).

Types of Access Lists

The access list can be one of two types:

- **PASN-AL** — the access list that is associated in z/VSE with a **partition**.
- **DU-AL** — the access list that is associated with a z/VSE **task** (main task or subtask).

A program uses the DU-AL associated with its z/VSE task and the PASN-AL associated with its partition.

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 partition, it might reference 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 partition, the program can share the data space with other programs running in the same partition.

If your program is not part of a subsystem, it will probably place entries for data spaces in its DU-AL.

Each z/VSE task has one DU-AL; all programs running under a z/VSE task can use it. A DU-AL cannot be shared with another z/VSE task. A program can, however, use the ALCOPY parameter on the ATTACH 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 96 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 data spaces that were represented by the entries on the DU-AL at the time of the attach.

Each partition has a PASN-AL. All programs running in the partition can use the PASN-AL of that partition. They cannot use the PASN-AL of any other partition.

The following lists summarize the characteristics of DU-ALs and PASN-ALs.

- The DU-AL has the following characteristics:
  - Each z/VSE task has its own unique DU-AL.
  - All programs running under a z/VSE task can add and delete entries on the z/VSE task's DU-AL.
  - A program cannot pass its task's DU-AL to a program running under another task. The one exception is that a program can pass a copy of its DU-AL to a subtask to be attached.
  - A DU-AL can have up to 253 entries.
- The PASN-AL has the following characteristics:
  - Every partition has its own unique PASN-AL.
  - Programs in PSW key 0 running in this partition can add and delete entries on the PASN-AL. A PASN-AL is also updated if a SCOPE=COMMON entry for a data space is added or deleted within the system.
  - All programs running in this partition can access data spaces through the PASN-AL.
  - A PASN-AL can have up to 253 entries, some of which are reserved for SCOPE=COMMON data spaces and virtual disks.

Because DU-ALs belong to z/VSE tasks, you must remember the relationship between the program and the z/VSE task under which the program runs. 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 task under which your program runs".

A Comparison of a PASN-AL and a DU-AL

Figure 19 on page 71 shows PGM1 that runs in partition 1. The figure shows the PASN-AL of partition 1 and PGM1's DU-AL. PGM1 shares the PASN-AL with other programs that execute in partition 1. It does not share its DU-AL with any other programs. The PASN-AL contains entries to data spaces that program(s) with PSW key 0 placed there. PGM1 has an entry for space X in its DU-AL and an ALET for space X. PGM1 received an ALET for space Y from a program running with PSW key 0. 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 provided the PSW key matches the storage key of space Y or the PSW key is 0.

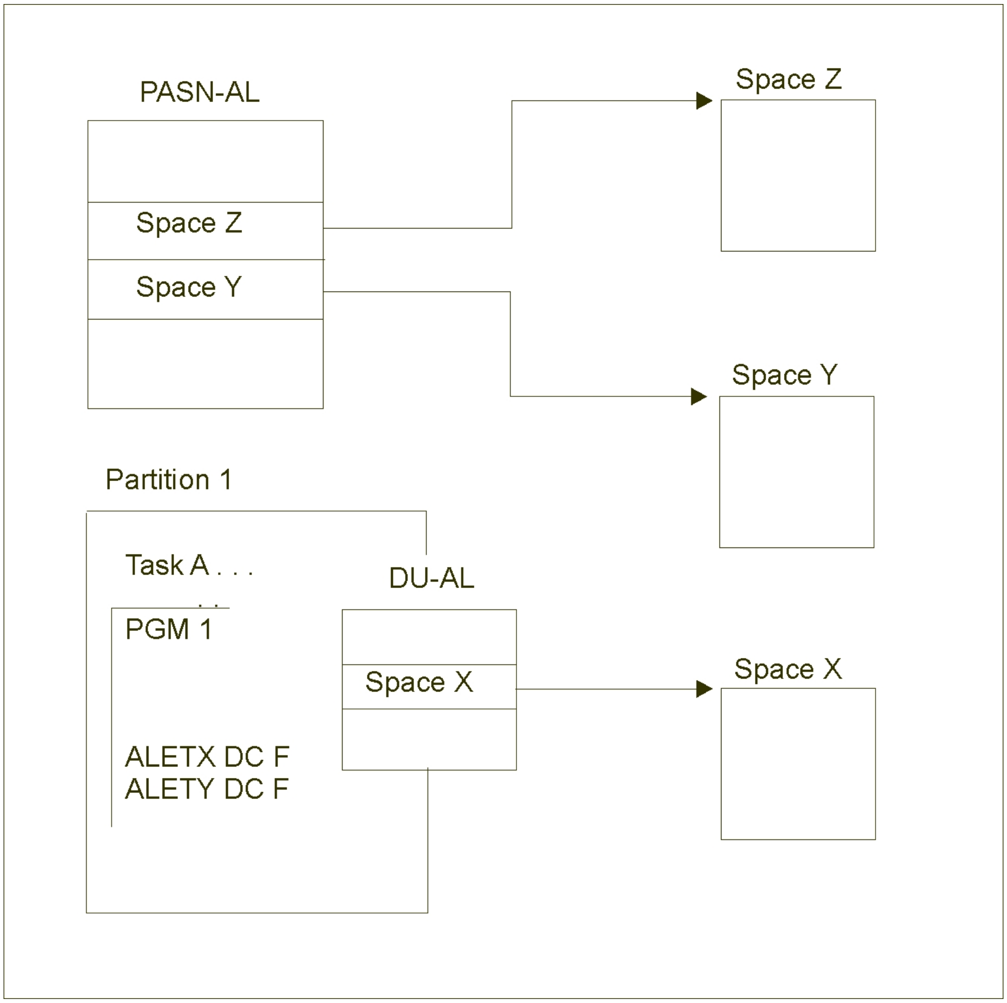


Figure 19. Comparison of Addressability through a PASN-AL and a DU-AL

Loading the Value of Zero into an AR

When the code you are writing is in AR mode, you must be very conscious of the contents of the ARs. For instructions that reference data, the ARs **must** always contain the ALET that identifies the data space that contains the data. 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 data spaces. Table 4 on page 72 lists some of the functions of the macro, the parameter that provides the function, and the section where the function is described.

Table 4. 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 73.
Delete an entry from an access list.	DELETE	"Deleting an Entry from an Access List" on page 78.
Obtain the STOKEN of a data space, given the ALET.	EXTRACT	"Obtaining and Passing ALETs and STOKENs" on page 74.
Find an ALET on an access list, given the STOKEN.	SEARCH	"Adding an Entry to an Access List" on page 73.

You can also find examples of the ALESERV macro in Chapter 10, "Creating and Using Data Spaces," on page 81.

Setting Up Addressability to a Data Space

Before your program can use ARs to reference data in a data space, it must establish a connection to the data space. The important facts to remember about setting up an environment in which your program can use ARs are in the following box.

Establishing addressability to a data space means that your program must:

- Have authority to access data in the data space
- Have an access list entry that points to the 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 data space, depends on whether the entry is for a data space. Authority to add an entry for a data space follows certain rules that are summarized in Table 5 on page 85. This table tells what PSW key 0 programs or programs with a non-zero PSW key can do with data spaces.

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 data space represented by the entry. You might have received the STOKEN from DSPSERV, or from another program.
- ALET — index to the entry that ALESERV added to the access list. The system returns this value at the address you specify on the ALET parameter.

An optional parameter, AL, allows you to limit access to a 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 data space to programs running under the owning z/VSE task. Use AL=PASN if you want other programs running in the same partition to have access to the data space, or if you are adding an entry for a SCOPE=COMMON data 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 94 describes the ALESERV ADD process for these data spaces.

ALESERV ADD is the **only** way to add an entry for a data space 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 73.
- “Examples of Establishing Addressability to Data Spaces” on page 75.

If you want to know whether a 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 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
         .
         .
*  SWITCH TO ACCESS REGISTER MODE
         .
         .
         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 93.

Obtaining and Passing ALETs and STOKENs

A program can obtain an ALET through the ALESERV macro with the ADD parameter. Or, it can receive an ALET from another program.

A program can obtain a STOKEN through DSPSERV CREATE, ALESERV EXTRACT. 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. z/VSE has certain rules for passing ALETs, as described in "Rules for Passing ALETs" on page 74. It does not have rules for passing STOKENs. However, the ALESERV service determines whether the receiving program can add an entry for the data space that a STOKEN represents.

Rules for Passing ALETs

z/VSE allows your program to pass the following ALETs:

- An ALET of zero (primary address space).
- An ALET that indexes into an entry on a DU-AL, if the program that passes the ALET and the program that receives the ALET run under the same z/VSE task (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 partition (that is, they have the same PASN-AL).
- An ALET that indexes into the PASN-AL for a SCOPE=COMMON data space.

To provide addressability to a data space, a program might pass an ALET to another program for which examples are provided below.

Examples of Establishing Addressability to Data Spaces

Note: Programs running in non-zero PSW key can access a data space for update only if storage and PSW key match.

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 partitions can access the same data space.

In these examples, programs share their data spaces with programs running under z/VSE tasks other than their own.

Example 1: Getting Addressability through a DU-AL

Consider that a PSW key zero 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 20 on page 75 shows PGM1 with the entry for data space DS1 on its DU-AL. It shows the STOKEN and the ALET.

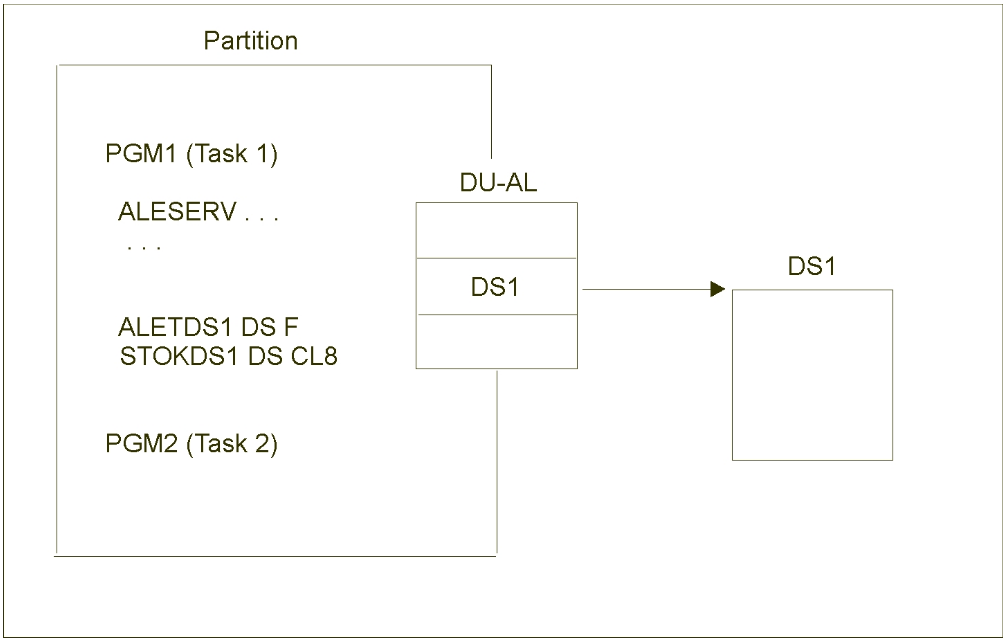


Figure 20. Example 1: Adding an Entry to a DU-AL

Consider that PGM2, with PSW key 0 and running under a task different from PGM1's task, would also like to have access to data space DS1. PGM1 passes PGM2 the STOKEN for DS1. PGM2 then uses the ALESERV ADD macro to obtain the ALET and add the entry. Figure 21 on page 76 shows PGM2 with addressability to DS1.

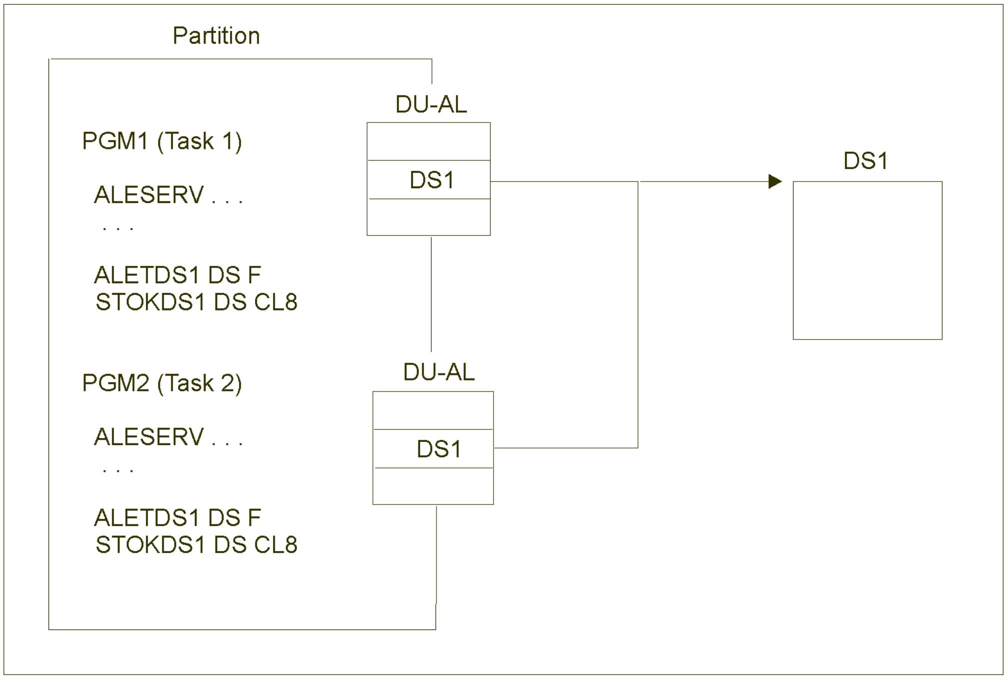


Figure 21. Example 1: Sharing a Data Space through DU-ALs

Note: A problem state program with PSW key unequal to zero cannot add entries to its PASN-AL, nor can the program add an entry on its DU-AL for a data space that was created by another task.

Example 2: Getting Addressability through a PASN-AL

In Figure 22 on page 77, consider that PROG1, with PSW key 0, 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 22 on page 77 shows PROG1 with the PASN-AL entry for data space DS2.

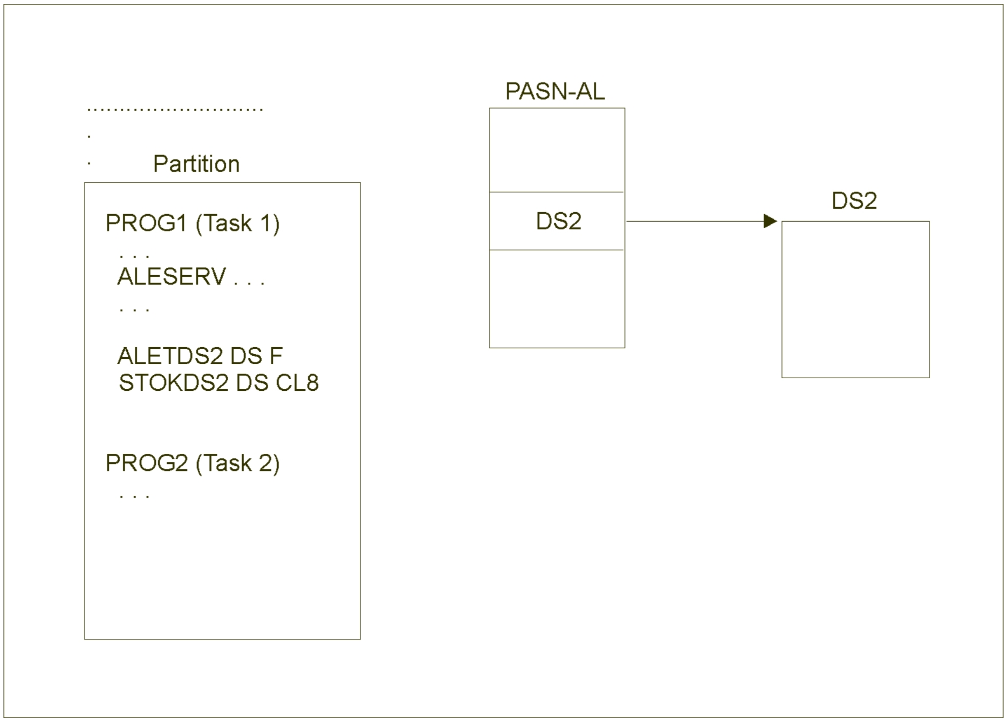


Figure 22. Example 2: Adding an Entry to a PASN-AL

Consider that PROG2 running under a task different from PROG1's would like to have access to data space DS2. In this case, both PROG1 and PROG2, because they run in the same partition, 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 23 on page 77 shows that PROG2 has the ALET for DS2 and, therefore, has addressability to DS2 through its PASN-AL.

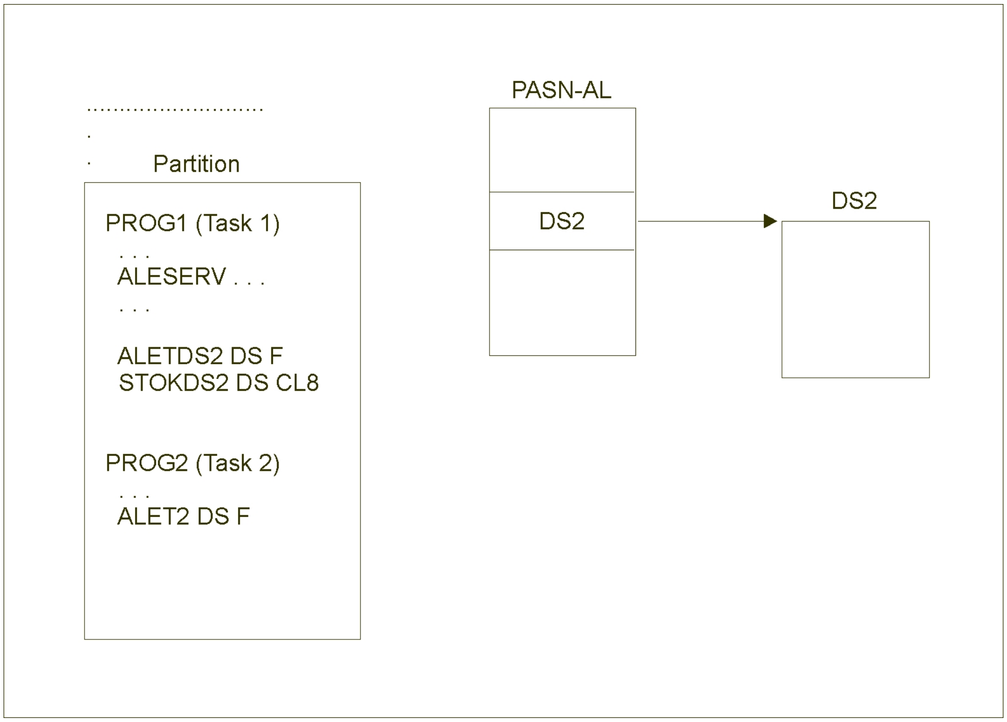


Figure 23. Example 2: Sharing a Data Space through the PASN-AL

In a similar way, any supervisor state or problem state program that runs in the same partition 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 82.

Example 3: Passing ALETs across Partitions

Referring to Figure 23 on page 77, consider that PROG1 wants to allow a program in another partition (same or different address space) to access data in data space DS2. Figure 24 on page 78 shows that PROG1 passes the STOKEN for DS2 to PROG2, a PSW key 0 program in Partition 2. PROG2 uses the ALESERV macro to add the entry to its DU-AL. PROG2 also could have added the entry to its PASN-AL.

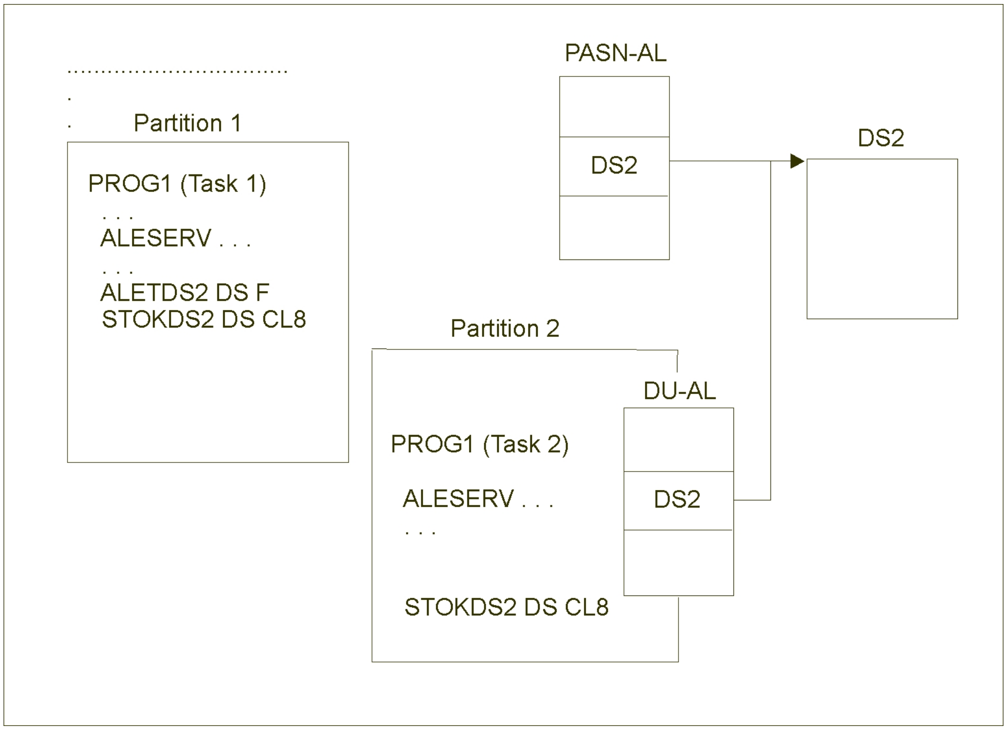


Figure 24. Example 3: Sharing Data Spaces Between two Partitions

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. Both, the DU-AL and the PASN-AL have 253 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:

- The system deletes a PASN-AL when a partition is de-allocated, a DU-AL when a task is terminated.
- Once the entry is deleted, the system can immediately reuse the access list entry.

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 79 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 z/VSE task terminates. At that time, the system frees the access list entry.

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 space or data space. When ALESERV DELETE removes an access list entry, the connection between the ALET and the data space no longer exists. The access list entry and its corresponding ALET are available for the system to be used 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 that other programs in the address space were also using ALETA to access DSA. For this reason, PROGA should tell those programs about the removal of ALETA to ensure that no access errors occur.

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 GETVIS storage that it obtained and shared with other programs. When that area of storage is freed, z/VSE reuses the area to satisfy a later request for storage. When an access list entry is freed, z/VSE reuses that ALET to satisfy a later ALESERV ADD request.

Chapter 10. Creating and Using Data Spaces

The support of data spaces is based on the following command and macros:

- SYSDEF command This JCL/AR command defines the size of the virtual storage available for data spaces. This size also depends on the values defined in the IPL VSIZE and DPD commands.
- ALESERV macro
- DSPSERV macro
- SYSSTATE macro
- SDUMPX macro

For overview information you may refer to "Macro and Command Support for Data Spaces" on page 135.

The DSPSERV macro 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.

A program's ability to create, delete, and access data spaces depends on whether it is a program with a non-zero PSW key or a PSW key 0 program. All programs can create, access, and delete the data spaces they own and share their data spaces with their subtasks. In addition, PSW key 0 programs can share their data spaces with other programs.

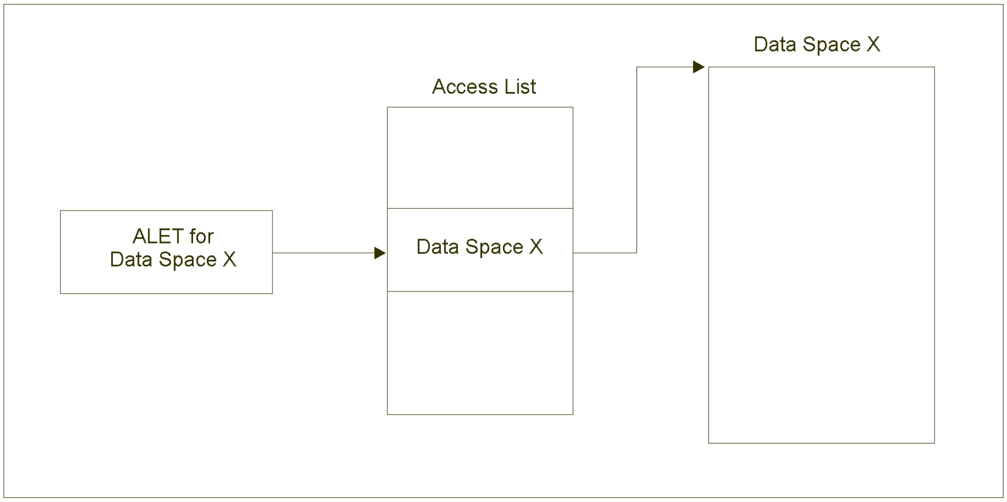
Use this topic to help you create, use, and delete data spaces. In addition, two sources of information can help you understand how to use data spaces:

- Chapter 9, "Using Access Registers," on page 63 contains many examples of setting up addressability to data spaces.
- *z/Architecture 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 IDs like address spaces). 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:

- DU-AL : the access list that is associated with a z/VSE main task or subtask.
- PASN-AL : the access list that is associated with a z/VSE partition.

Relationship Between the Data Space and Its Owner

The owner of a data space is a z/VSE main task or subtask. When a task terminates, the system deletes any data spaces that the task owns.

This section 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 task 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 partition. A SCOPE=SINGLE data space with an entry on a DU-AL can only be used by the creating task.
- **SCOPE=ALL** data spaces

  A SCOPE=ALL data space can be used by programs running in the owner's partition or other partitions. SCOPE=ALL data spaces provide a way to share data selectively among programs running in different partitions.
- **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 shared virtual area (SVA).

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 programs with non-zero PSW key differ from the rules for programs that are PSW key 0. The table on page Table 5 on page 85 summarizes these rules and the example on Figure 25 on page 84 illustrates them.

A program with PSW key 0 can:

- **Create** a data space.

- **Delete** a data space if any task of the caller's partition owns the data space.
- **Release** storage in a data space if any task of the caller's partition owns the data space.
- **Extend the current size** of any data space it owns if any task of the caller's partition owns the 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.
  - For a SCOPE=SINGLE data space to its PASN-AL, if the PASN-AL belongs to the owner's partition.
  - For any SCOPE=ALL data space to its DU-AL and its PASN-AL.
  - For any SCOPE=COMMON data space to its PASN-AL.

  Note that programs with non-zero PSW key cannot add entries to their PASN-ALs. PSW key 0 programs, however, can add entries on behalf of programs with non-zero PSW key and pass copies of the ALETs that index the entries to these 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 if it runs under the same task which owns the DU-AL. 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 25 on page 84 shows two partitions (Partition 1, Partition 2) and two data spaces (DS1, DS2). The entries in the PASN-AL and DU-AL are identified.

Two programs run in Partition 1, both of which own data spaces:

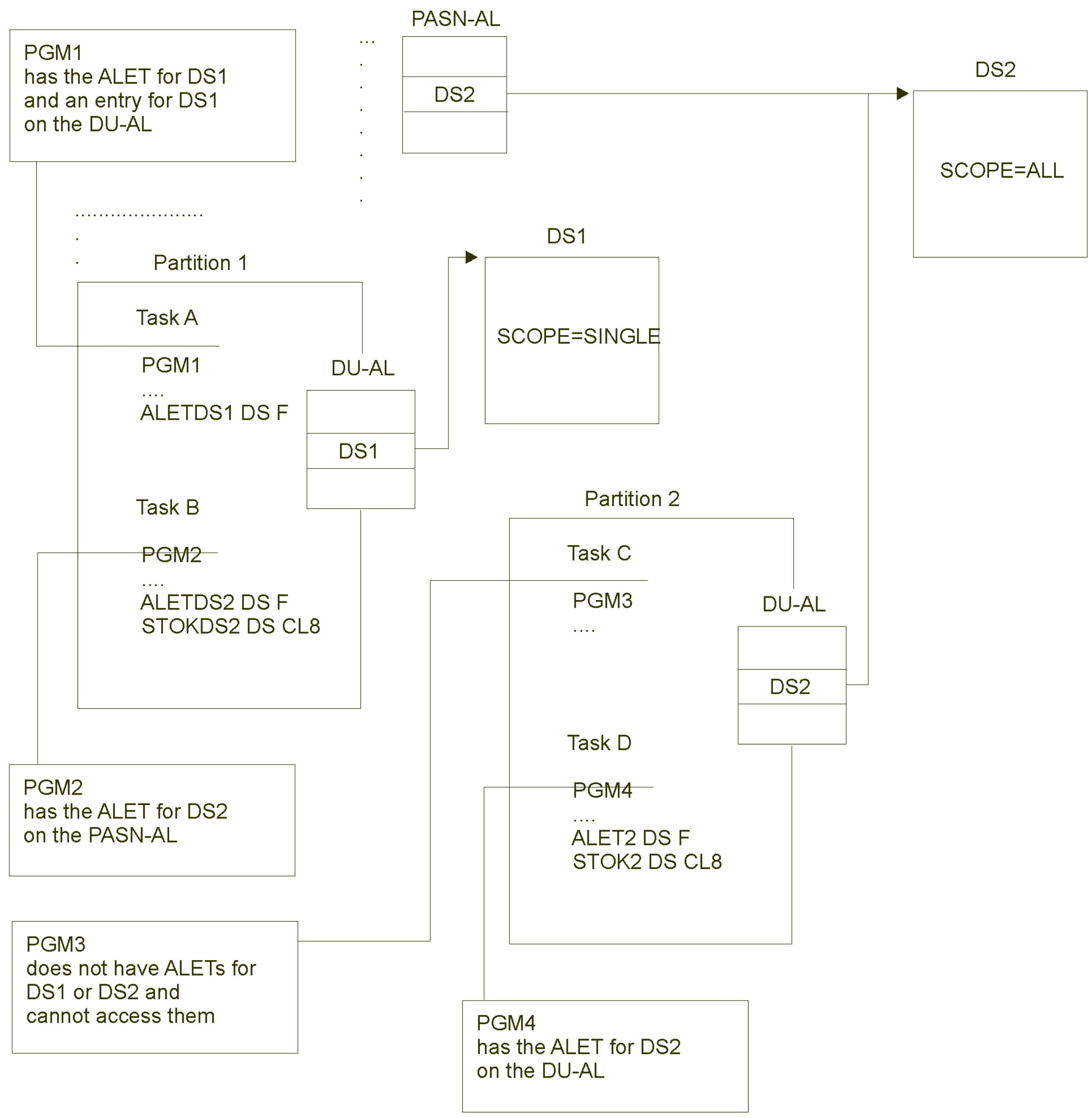
- A program with non-zero PSW key, PGM1, running under Task A that owns the SCOPE=SINGLE data space DS1.
- A PSW key 0 program, PGM2, running under Task B that owns the SCOPE=ALL data space DS2.

Two programs run in Partition 2, neither of which owns data spaces:

- A program with non-zero PSW key, PGM3, running under Task C.
- A PSW key 0 program, PGM4, running under Task D.

PGM2 has passed a STOKEN for the SCOPE=ALL data space DS2 to PGM4 in Partition 2. 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 section, it was stated that storage within a data space is available to programs that run under the task that owns the data space. The exception to this statement is when the owning task has the data space entry on the PASN-AL.



The four programs have the following relationship to the two data spaces:

- PGM1 can access DS1 through its DU-AL. Because it runs in non-PSW key 0. PGM1 cannot add an entry for DS1 to the PASN-AL.
- 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.)
- PGM3 cannot access either DS1 or DS2.
- PGM4 can access DS2 through its DU-AL.

Figure 25. Example of Rules for Accessing Data Spaces

Summary of Rules for Creating, Deleting, and Using Data Spaces

Table 5 on page 85 summarizes the rules for what programs can do with data spaces. The third column describes what a program with non-zero PSW key can do. The fourth column describes what a PSW key 0 can do.

Table 5. Creating, Deleting, and Using Data Spaces

Function	Type of data space	A program with non-zero PSW key:	A program with PSW key 0:
CREATE	SCOPE=SINGLE	Can create a SCOPE=SINGLE data space.	Can create the data space.

	SCOPE=ALL
	SCOPE=COMMON
		Cannot create the data spaces.	Can create the data space.

DELETE	SCOPE=SINGLE	Can delete the data spaces it owns if its PSW key matches the storage key of the data space.	Can delete the data space if any task of the caller's partition owns the data space.

	SCOPE=ALL
	SCOPE=COMMON
		Cannot delete the data space.	Can delete the data space if any task of the caller's partition owns the data space.

RELEASE	SCOPE=SINGLE	Can release storage in data spaces it owns if its PSW key matches the storage key of the data space.	Can release storage in a data space if any task of the caller's partition owns the data space.

	SCOPE=ALL
	SCOPE=COMMON
		Cannot release the storage.	Can release storage in a data space if any task of the caller's partition owns the data space.

EXTEND	SCOPE=SINGLE	Can extend storage in data spaces it owns.	Can extend storage in a data space if any task of the caller's partition owns the data space.

	SCOPE=ALL
	SCOPE=COMMON
		Cannot extend the current size.	Can extend the current size if any task of the caller's partition owns the data space.

Add or delete entries in the DU-AL	SCOPE=SINGLE	Can add or delete entries for the data spaces it owns.	Can add or delete entries for the data space if any task of the caller's partition owns the data space.

	SCOPE=ALL
	SCOPE=COMMON
		Cannot add or delete entries.	Can add or delete entries for a SCOPE=ALL (not the SCOPE=COMMON) data space.

Add or delete entries in the PASN-AL	SCOPE=SINGLE	Cannot add or delete entries.	Can add or delete entries for a data space if its PASN-AL is the same as the PASN-AL of the owner's partition.

	SCOPE=ALL
	SCOPE=COMMON
		Cannot add or delete entries.	Can add or delete 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 also 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.

Creating a Data Space

To create a data space, issue the **DSPSERV CREATE** macro. z/VSE 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 a Data Space” on page 86.
- 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 macro.

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 82.
- 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 also “Specifying the Size of the Data Space” on page 86.

Choosing the Name of a Data Space

The name you specify on the NAME parameter will identify the data space on dump requests and AR/JCL commands.

Names of data spaces must be unique within a partition. 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 z/VSE uses, z/VSE has some specific rules for you to follow. Refer to the manual z/VSE System Macros Reference under "DSPSERV CREATE (Create Data Space) Macro" for details.

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 4KB is 2GB.) The addressing range for the data space depends on the processor. Before you code BLOCKS, you should know two facts about how an installation controls the use of virtual storage for data spaces.

- An installation can set limits on the amount of storage available for all data spaces (through the SYSDEF command). 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 (through the SYSDEF command). If you do not use the BLOCKS parameter of the DSPSERV CREATE macro or the BLOCKS parameter is zero, the system creates a data space with the default size. To find out about data space sizes, use the QUERY DSPACE command.

For information on how to change IBM defaults, see “Limiting Data Space Use” on page 89.

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.

Note: The storage taken from VSIZE is either the initial or extended data space size rounded up to the next multiple of 8 BLOCKS.

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 storage, the system uses the current size of the data space.

If you know what data space size 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 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 returns the maximum 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 26 on page 87 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.

```
          DSPSERV CREATE,. . .BLOCKS=(DSPMAX,DSPINIT)
          .
          .
DSPMAX  DC  A((100000+4095)/4096)         DATA SPACE MAXIMUM SIZE
DSPINIT DC  A((20000+4095)/4096)          DATA SPACE INITIAL SIZE
```

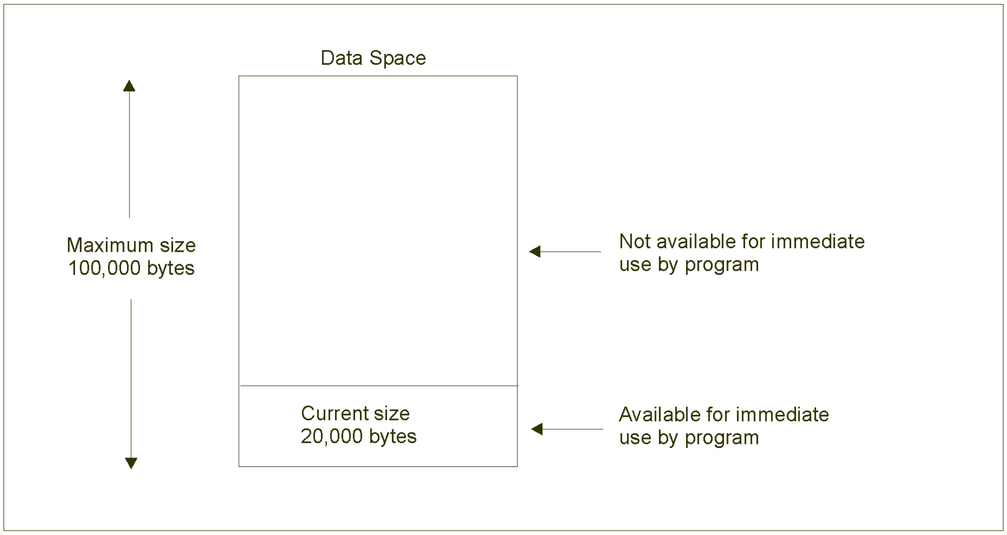


Figure 26. Example of Specifying the Size of a Data Space

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 92 describes extending the current size and includes an example of how to extend the current size of the data space in Figure 26 on page 87.

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. 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 start address of the data space at the location you specify.

Unless you specify a size of 2GB 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 1MB 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.

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 section; section Chapter 9, “Using Access Registers,” on page 63 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. It is recommended to use multiples of 8 BLOCKs for these values.
- If you do not use an area of a data space again, release that area to free the resources occupied by the area.

  Refer also to “Releasing Data Space Storage” on page 97 for additional details.
- When you are finished using a data space, remove its entry from the access list and delete the data space.

  Refer also to “Deleting a Data Space” on page 93 for additional details.

Limiting Data Space Use

The use of data spaces consumes system resources such as virtual storage (and real storage). Programmers responsible for tuning and maintaining z/VSE can control the use of these resources. Through the SYSDEF command you can specify the maximum amount of virtual storage available for all data spaces.

You may also specify:

- The default size of a single data space.
- The number of data spaces that can be owned by a single partition.
- The maximum number of SCOPE=COMMON data spaces.

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 LOCK and UNLOCK, the ENQ and DEQ macros, compare and swap operations, or establish your own protocol for serializing data space use.

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 27 on page 90 shows a SCOPE=ALL data space DSX with a storage key of 5, owned by a subsystem. PGM1 and PGM2 have entries for the data space on their DU-ALs and have the ALETs for these entries. However, the PSW key of PGM1 does not match the storage key of the data space. The ability of PGM1 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 can fetch from but not store into the data space.
- If the creating program specified fetch-protection (FPROT=YES), PGM1 can neither fetch from nor store into the data space.

Figure 27 on page 90 shows that only PGM2 has fetch and store capability for data space DSX.

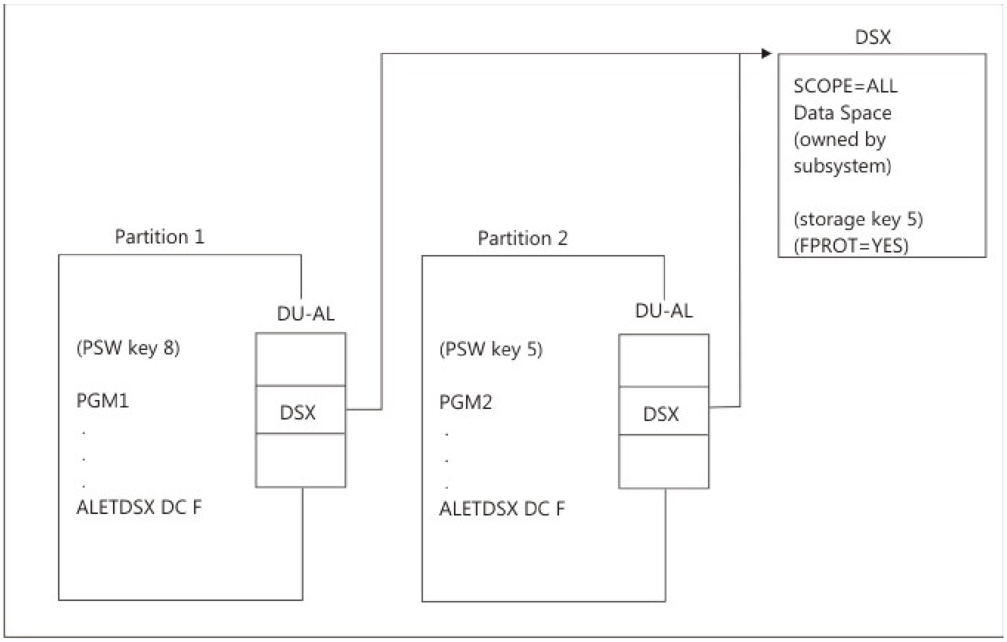


Figure 27. Protecting Storage in a Data Space

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
```

For establishing addressability, refer also to the coding example under "Example of Creating, Using, and Deleting a Data Space" on page 93.

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
         ....                        SAVE CALLER'S STATUS
         LAE   12,0(0,0)             BASE REG AR
         BASR  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
         BR    ...                   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
         ....                        SAVE CALLER'S STATUS
         LAE   12,0(0,0)             BASE REG AR
         BASR  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
         BR    ....                  RETURN TO CALLER
```

Programming Notes for 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 2^{24}-1 bytes (or 16,777,215 bytes).

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 amount of virtual storage available for all data spaces. This limit is the system default or is set by the AR/JCL SYSDEF command.

Note: The storage taken from VSIZE is either the initial or extended data space size rounded up to the next multiple of 8 BLOCKS.

Consider the data space in Figure 26 on page 87, 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 28 on page 92 :

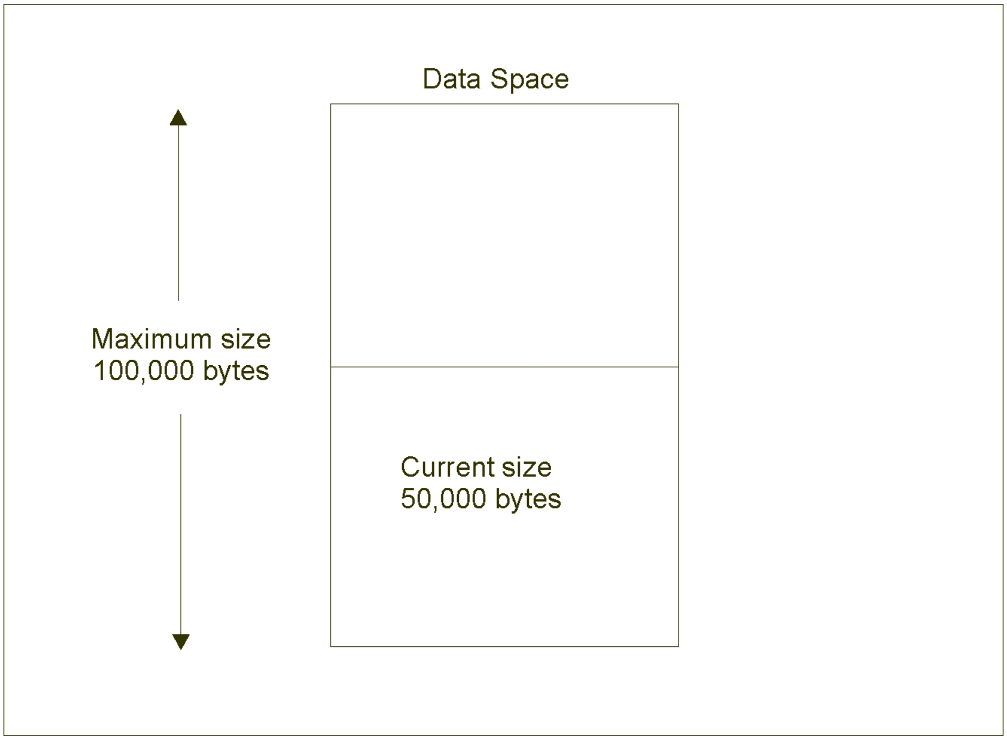


Figure 28. Example of Extending the Current Size of a Data Space

Deleting a Data Space

When a task does not need the data space any more, it can free the virtual storage and remove the entry from the access list.

A program with a non-zero PSW key can delete only the data spaces it owns, and must have the PSW key that matches the storage key of the data space.

Example of Deleting a Data Space

The following example shows 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. However, if you do not, z/VSE automatically does it for you.

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

         ....                          SAVE CALLER'S STATUS
         SAC   512                     SWITCH INTO AR MODE
         SYSSTATE ASCENV=AR            SET GLOBAL BIT FOR AR MODE
         .
*  SET UP AR/GPR 12 BEFORE STORING INTO IT
         .
         LAE   12,0                    SET BASE REGISTER AR
         BASR  12,0                    SET BASE REGISTER GR
         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                LOAD DATA FROM PRIMARY SPACE
         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
         .
         SAC   0                         ENSURE IN PRIMARY MODE
         SYSSTATE ASCENV=P               RESET GLOBAL BIT
         BR   ...                        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: 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 shared virtual area (SVA) of an address space. You might use a SCOPE=COMMON data space instead of SVA because:

- A SCOPE=COMMON data space offers up to 2GB of commonly addressable virtual storage for data (but not executable code). The SVA offers a much smaller amount of storage.
- The SVA is a limited resource; because it is a part of all address spaces, the use of this virtual storage area reduces the amount of private area available for partitions (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 SVA. A program wanting to share SVA 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 29 on page 95 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 is deleted. PROG1 has the ALET for the entry.

To allow programs in other partitions to access data space COMDS, PROG1 can pass the ALET to these other programs.

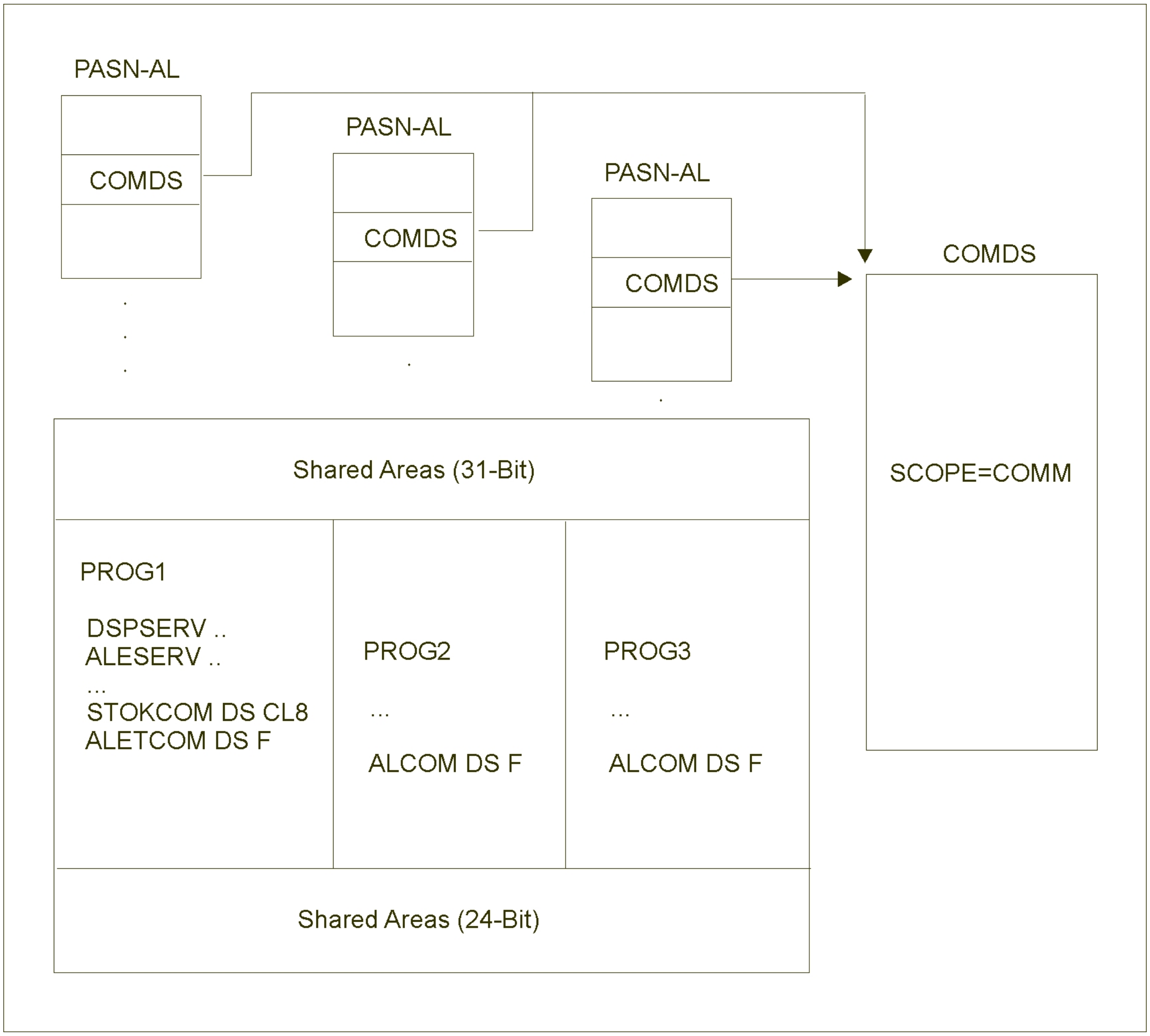


Figure 29. Example of Using a SCOPE=COMMON Data Space

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 10MB of commonly addressable data, consider using a SCOPE=COMMON data space. To use less than 10 MB, consider using SVA (31-Bit).
- 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 79.
- 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 and Displaying Data Space Storage" on page 97 for information about dumping data space areas.

Note: Your installation can set limits on the total number of SCOPE=COMMON data spaces available to programs.

The maximum number is 253, minus the number of virtual disks specified. The minimum number is 5, and the default is also 5. These numbers include the SCOPE=COMMON data spaces that the system has for its own use.

When you set this limit, remember that the number it establishes affects the number of PASN-AL entries that are available for data spaces other than SCOPE=COMMON. The number of entries each PASN-AL has available for SCOPE=SINGLE and SCOPE=ALL data spaces is 253 minus the number you set for SCOPE=COMMON minus the number of virtual disks specified.

Attaching a Subtask and Sharing Data Spaces with It

A program can use the ALCOPY=YES parameter on the ATTACH 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 with a program running under the subtask. The two programs both have access to the data spaces that have DU-AL entries at the time of the ATTACH macro invocation. Note that it is not possible to pass only a part of the DU-AL.

The following example, represented by Figure 30 on page 96, assumes that program PGM1 (running under Task A) 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 ATTACH macro to attach Subtask B (Task B). PGM1 uses the ALCOPY=YES parameter to pass a copy of its DU-AL to Task B. It can also pass ALETs to PGM2. Upon return from ATTACH, PGM1 and PGM2 have access to the same data spaces.

The figure shows the two programs, PGM1 and PGM2, sharing the same data space, DS1.

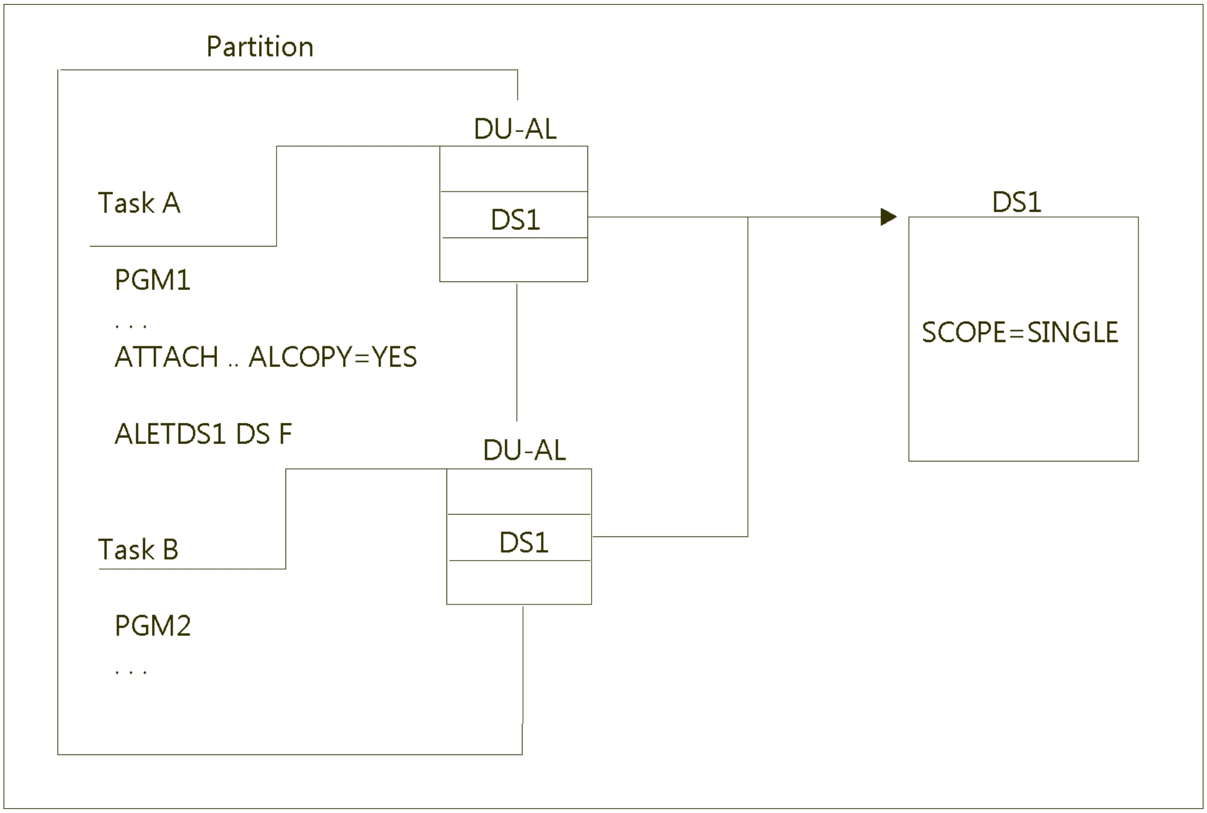


Figure 30. Two Programs Sharing a SCOPE=SINGLE Data Space

Example of Attaching a Task and Passing a DU-AL

The following example shows you how Task A attaches Subtask B and passes its DU-AL:

```
   DSPSERV CREATE,NAME=DSNAME,BLOCKS=DSSIZE,STOKEN=DSSTOK,ORIGIN=DSORG
   ALESERV ADD,STOKEN=DSSTOK,ALET=DSALET
   ATTACH 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 Task A terminates, the system deletes the data space that belonged to Task A. PGM2 can no longer access DS1.

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.

Releasing storage in a data space requires that a program's PSW key to be zero or equal to the key of the data space storage the system is to release. Otherwise, the system abends 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.

Using Data Spaces Efficiently

Although a task can own many data spaces, it is important that it references 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, a CICS application might have 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.

z/VSE limits the number of access list entries and the number of data spaces available to each task. Therefore, IBM recommends that, given a choice, you 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.

Dumping and Displaying Data Space Storage

z/VSE provides various ways to dump areas of data space storage. You can use the SDUMPX macro or ask the operator to use the DUMP command. After the system enters a wait state or hangs or enters a loop, the operator can request a stand-alone dump.

Use the following to dump data space storage:

1. Use the **SDUMPX** macro to dump storage from any data space that the caller has addressability to.
2. An operator can use the DSPACE parameter on the **DUMP** command to dump all the storage of a data space.

   The operator can use the QUERY command to determine the names of the data spaces owned by a partition or the names of all data spaces of the system.
3. An operator can request a stand-alone dump as described in the z/VSE Guide for Solving Problems under "Taking a Stand-Alone Dump" .

The following options support the dumping of data space storage:

- // OPTION statement
  - Use **DSPDUMP** to specify that a data space dump is to be taken in case of an abnormal program end.
  - Use **SADUMP** to specify the data spaces to be included in a stand-alone dump.

- For the standard options (STDOPT) you can specify **DSPDUMP** to specify that a data space dump is to be taken in case of an abnormal program end.

  In addition, it is possible to specify SADUMP to define the priority in which the partitions and any owned data spaces should be included in a stand-alone dump.

For further details about these options, refer to z/VSE System Control Statements.

Chapter 11. Creating and Using Virtual Disks

On an z/Architecture processor, z/VSE supports data spaces of up to 2GB. *Virtual disks* are based on this support. Such disks emulate real FBA disk devices and allow data that normally would be stored on a real disk device to reside in memory (virtual storage). Since such data can be accessed at memory speed, response time and throughput in general may improve significantly.

Planning for Virtual Disks

Since virtual disks are not permanent, they should be used for files that easily can be recovered in case of loss (because of an IPL, for instance). These include, for example:

- **Temporary work files or test files**
- **VSE/VSAM space and user catalogs**
- **VSE libraries**

Before you can use virtual disks, you need to allocate virtual storage for data spaces in your system layout. You do this by adjusting the IPL parameter **VSIZE**. If no other data spaces are needed besides those for virtual disks, then add the size of all virtual disks that might be used *concurrently* to the value of VSIZE.

Once you have determined the value for VSIZE, also use this value to adjust the size of your page data set (defined through the IPL command **DPD**).

Creating Virtual Disks

You can easily create virtual disks by:

1. Updating your IPL procedure with an IPL **ADD** command for each virtual disk.
2. Using the AR **SYSDEF** or JCL **SYSDEF** command from the BG partition to specify the total amount of space used by all data spaces (including those for virtual disks).
3. Issuing a JCL **VDISK** command from the BG partition for each virtual disk.

Note:

1. You may use the *Tailor IPL Procedure* dialog for adding ADD commands. The dialog is described in z/VSE Administration.
2. You may include the VDISK command in the JCL startup procedure for the BG partition. The skeleton to be used for this modification is SKJCL0. The SYSDEF command is included in the ALLOC startup procedure and can be modified with one of the SKALLOCx skeletons. All skeletons are described in the z/VSE Administration manual.
3. You can define up to 128 virtual disks.

Following is an overview of the commands mentioned above. For further command details, refer to z/VSE System Control Statements.

ADD Command

As for real disks, you need to define your virtual disks by ADD commands at system IPL. The ADD command has the following operands:

cuu
Specifies the device address of the virtual disk. This can be any unused cuu of your system.

FBAV
Indicates that the device is a virtual disk with FBA characteristics.

SYSDEF Command

The SYSDEF command defines limits and defaults for data spaces. In particular, it defines the amount of virtual storage which may be allocated to data spaces if available and not used by partitions. It must be large enough to accommodate all concurrently used virtual disks, plus any other data spaces. This virtual storage is part of the virtual storage defined with the VSIZE parameter at IPL. The SYSDEF command has the following operands:

DSPACE
Indicates a request for virtual storage used by data spaces.

DSIZE=nK|mM
Defines the total amount of virtual storage to be reserved for data spaces.

Note: **0K** or **0M** can be used to free up all data space allocations.

MAX=n
Defines the maximum number of data spaces which may be allocated.

PARTMAX=m
Defines the maximum number of data spaces which may be created by a partition at any one time.

COMMAX=k
Defines the maximum number of data spaces with SCOPE=COMMON which may exist at any one time.

DFSIZE=nK|mM
Defines the default size for the creation of a single data space.

VDISK Command

For *every* virtual disk needed, you must issue a VDISK command from the BG partition. This command creates the data space used for the virtual disk and a VTOC that indicates an empty disk.

After the command completes, the virtual disk is initialized and ready for use. You do not have to initialize the disk using the ICKDSF (Device Support Facilities) program. The VDISK command has the following operands:

UNIT=cuu
Indicates the device address of the virtual disk.

BLKS=nnnnnnn
Defines the size of the virtual disk in 512-byte blocks.

This value is always rounded to a multiple of 960, as described in z/VSE System Control Statements.

Note: For information about redefining the space allocated for an existing virtual disk, see "Deleting or Redefining Virtual Disks" on page 101.

VOLID=volser
Specifies the volume serial number (one to six characters) of the virtual disk.

VTOC=v
Defines the number of blocks to be used for the VTOC. The VTOC is located at the end of the virtual disk.

USAGE=DLA
Indicates that this virtual disk is to hold the label information area. The z/VSE Planning explains how the label area is placed on a virtual disk at IPL time.

Note: Once a virtual disk is defined and ready for use, you can work with it like a real device. This includes using the following job control commands and statements: ASSGN, CLOSE, DVCDN, DVCUP, OFFLINE, and ONLINE. ASSGN and CLOSE now also allow a device-class specification of FBAV (or ANYFBA or ANYDISK) for virtual disks.

Defining a Virtual Disk via the Interactive Interface

The *Hardware Configuration* dialog of the Interactive Interface supports the device type FBAV for a virtual disk. The selection list for disk devices includes this device type.

The Interactive Interface dialogs do not support the following functions for virtual disks:

- Specify SWITCHED or SHARED.
- Create a stand-alone dump program.
- Define the lock file.
- Define the page data set.
- Define the recorder file.
- Define the VSE/VSAM master catalog.

Getting Information about Virtual Disks

VOLUME Command

At the system console, you can use the Attention Routine **VOLUME** command to display information about existing virtual disks. In the following example of the system's response to this command:

1. Parameter D0 in the VOLUME command is used to request information about devices with cuu=D0x.
2. The virtual disk at address D01 is ready for use. It was added during IPL and a subsequent VDISK command was used to define space for it. Thus the information for this device shows:
   a. **FBA0-00** as device type.
   b. **VDID01** as a system-generated volume ID.
   c. **4800** as the number of blocks specified in the VDISK command.
3. The virtual disks at addresses D02 through D05 only have been added at IPL. A VDISK command has not yet been used to allocate space to them. Here the DEVICE column shows **FBAV** as device type. **DOWN** (disk is turned off) is the status of these virtual disks.

```
VOLUME D0
AR 0015 CUU  CODE DEV.-TYP   VOLID  USAGE   SHARED    STATUS    CAPACITY
AR 0015 D01  90   FBA0-00   VDID01  UNUSED                       4800 BLK
AR 0015 D02  90E5 FBAV      *NONE*  UNUSED            DOWN          N/A
AR 0015 D03  90E5 FBAV      *NONE*  UNUSED            DOWN          N/A
AR 0015 D04  90E5 FBAV      *NONE*  UNUSED            DOWN          N/A
AR 0015 D05  90E5 FBAV      *NONE*  UNUSED            DOWN          N/A
AR 0015 1I40I  READY
```

Figure 31. Response Example to a VOLUME Command

For a description of the VOLUME command, refer to z/VSE System Control Statements.

QUERY DSPACE Command

To retrieve information about data spaces and related virtual disks, you can use the QUERY DSPACE command. For a description of the QUERY SPACE command, refer to z/VSE System Control Statements.

Deleting or Redefining Virtual Disks

Every time you shut down your system, you "delete" all virtual disks and the data they contain. As shown below, you also can effectively delete a disk by setting the number of blocks assigned to it to **0**. Later, you can reuse that virtual disk by reallocating space to it.

The following example assumes a virtual disk with device address 256. The VDISK command for that disk specified BLKS=960 (about 0.5MB). The first VDISK command in the example "deletes" the virtual disk by deallocating the space used for it. The second VDISK command redefines the disk with a larger allocation.

```
0 DVCDN 256                  (Makes the virtual disk no longer available to
                              the system)
0 VDISK UNIT=256,BLKS=0      (Deallocates previously defined space)

0 VDISK UNIT=256,BLKS=1920   (Redefines the space for the virtual disk;
                              DVCUP done implicitly)
```

Note: The DVCDN command can be issued from any static partition. VDISK must be used from the BG partition.

Programming Notes

Supported CCW Codes for Virtual Disks

The major CCW commands for accessing a real FBA disk device are also supported for virtual disks. These include:

- X'63' DEFINE EXTENT
- X'43' LOCATE
- X'42' READ
- X'41' WRITE
- X'03' NO OPERATION
- X'04' SENSE
- X'08' TRANSFER IN CHANNEL
- X'64' READ DEVICE CHARACTERISTICS
- X'E4' SENSE ID

Other commands that are valid for a real FBA device (X'94' DEVICE RELEASE, for example) are not supported for virtual disks. For further information about CCW commands, see Appendix C, “Channel Program Support for Virtual Disks,” on page 139.

GETVCE Macro

To retrieve the device characteristics of a virtual disk from an application, you can use the **GETVCE** macro, together with macro **AVRLIST**. The external device type code you receive for a virtual disk is **FBA000** for a virtual disk defined with the VDISK command. Otherwise, it is FBAV.

Part 4. Programming Enhancements

Chapter 12. Linkage Stack Functions

Linkage stack functions are available with z/Architecture processors and are documented in the following manual:

z/Architecture Principles of Operation

This manual describe in detail the linkage stack instructions introduced in this topic.

Introduction

The linkage stack is an area of protected storage that the system gives to a program to save status information in case of a branch or a program call.

Each VSE task has its own linkage stack, which is available for all programs running under this task. The programs can run in primary ASC (address space control) mode or in access register ASC mode. A program can be in problem or supervisor state and be enabled or disabled.

The linkage stack consists of two stacks, the

- normal linkage stack, and the
- recovery linkage stack.

The **normal linkage stack** consists of a maximum of 96 entries for use by programs that run under a single VSE task. 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.

The **recovery linkage stack** is available to the program's recovery routines after a "stack full" interruption occurred. The recovery linkage stack has a maximum of 24 entries.

Linkage Stack Characteristics

A linkage stack has the following characteristics.

- Each VSE task has its own linkage stack.
- All associated linkage stack entries of a VSE task are freed when the task is terminated.
- A program called by a // EXEC ... JCL statement will receive an empty linkage stack.
- The following instructions can be used to add and remove linkage stack entries and access their contents:

  BAKR, EREG, ESTA, MSTA, PC, PR

  These instructions can execute in both primary and access register ASC mode and are further explained in the following paragraphs.
- A **linkage stack entry** contains the following saved program status information:
  - Contents of the 16 (0-15) general purpose registers (GRPs).
  - Contents of the 16 (0-15) access registers (ARs).
  - Primary and secondary address space numbers (PASN and SASN).
  - Extended authorization index (EAX).
  - Entire current program status word (PSW).
  - PSW key mask (PKM).
  - PC (program call) number if a PC instruction caused the entry; or a return address if a BAKR instruction caused the entry.
  - An eight-byte area that can be changed with the MSTA (modify stacked state) instruction.

Instructions for Adding or Removing a Linkage Stack Entry

There are three instructions for adding or removing entries in a linkage stack:

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

The Stacking PC (Program Call) Instruction

The *stacking PC* uses the linkage stack to save the user's environment. The following restrictions apply:

- z/VSE supports the stacking PC for system provided program call (PC) routines only (it does not support the *basic PC*).
- z/VSE does not support the stacking PC in supervisor appendages.

The BAKR (Branch and Stack) Instruction

The BAKR instruction performs a branch and link in a similar way as the BALR instruction. The difference is that BAKR does not change the link register contents (the register contents will be used for the return via the PR instruction). Additionally, it adds an entry to the linkage stack. The entry includes the return address of the calling program. A program return (PR) instruction returns control to the calling program and removes the entry from the linkage stack. The BAKR instruction does not change the current addressing mode.

z/VSE does not support the BAKR instruction in most supervisor appendages.

The PR (Program Return) Instruction

The PR instruction performs several actions on the **current entry** in the linkage stack (the current entry is the entry created 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.
- It restores the contents of the current entry, including the PSW and contents of ARs and GPRs 2 through 14.
- It removes the current linkage stack entry.

Instructions to Work with Linkage Stack Entries and their Contents

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

Using the STXIT and EXIT Macro in Connection with Linkage Stack

A **STXIT** macro (AB, IT, OC, or PC exit) is allowed only if the linkage stack is empty. If a STXIT macro is issued in a module called with a PC or BAKR instruction (causing the linkage stack to be not empty), the calling task is canceled with cancel code X'47'.

If an AB exit completes with macro **EXIT** AB, the program continues with an empty linkage stack. For the other exits (IT, OC, or PC) the program continues with the linkage stack contents at the time of exit initiation.

When an exit routine returns control, the contents of the linkage stack must be the same as it was on entry of the exit routine, otherwise the exit routine is canceled with cancel code X'47'. To ensure that the contents is the same, an exit routine must use paired instructions (BAKR/PC - PR).

Chapter 13. Callable Cell Pool Services

Callable cell pool services manage virtual storage located in either an address space or a data space. The GETVIS macro or the DSPSERV macro can be used to obtain the virtual storage to be managed by cell pool services. Applications that use callable cell pool services must be compiled with the **High Level Assembler**.

Characteristics of a Cell Pool

Cell storage is an area of virtual storage that is subdivided into fixed-sized areas of storage called **cells**, where the cells are of the size you specify. To manage cell storage (shown in Figure 32 on page 110), cell pool services also require virtual storage for **anchor** and **extent** areas. Thus, a cell pool contains:

- An anchor
- At least one extent
- Any number of cells (all having the same size).

The **anchor** is the starting point or foundation on which you build a cell pool. Each cell pool has only one anchor. An **extent** contains information that helps callable cell pool services manage cells and provides information you might request about the cell pool. A cell pool can have up to 65,536 extents, each responsible for its own cell storage. Your program determines the size of the cells and the cell storage.

Note that the cell pool services manage a cell pool but do not access the cell storage itself.

The virtual storage for the cell pool must reside in an address space or a data space. The following requirements exist:

- The anchor and extents must reside within the same address space or data space.
- The cells must reside within one address space or data space; that space can be different from the one that contains the anchor and extents.

Figure 32 on page 110 illustrates the cell pool structure. In the example, the anchor and extents reside in Address or Data Space A and the cell storage in Address or Data Space B.

Before you can obtain the first cell from a cell pool, you must plan the location of the anchor, the extents, and the cell storage. You must obtain the storage for the following areas and pass the following addresses to the services:

- The anchor, which requires 64 bytes of storage.
- The extent, which requires 128 bytes plus one byte for every eight cells of cell storage.
- The cell storage.

Through the callable cell pool services, you build the cell pool. You can then obtain cells from the pool. When there are no more cells available in a pool, you can use callable cell pool services to enlarge the pool.

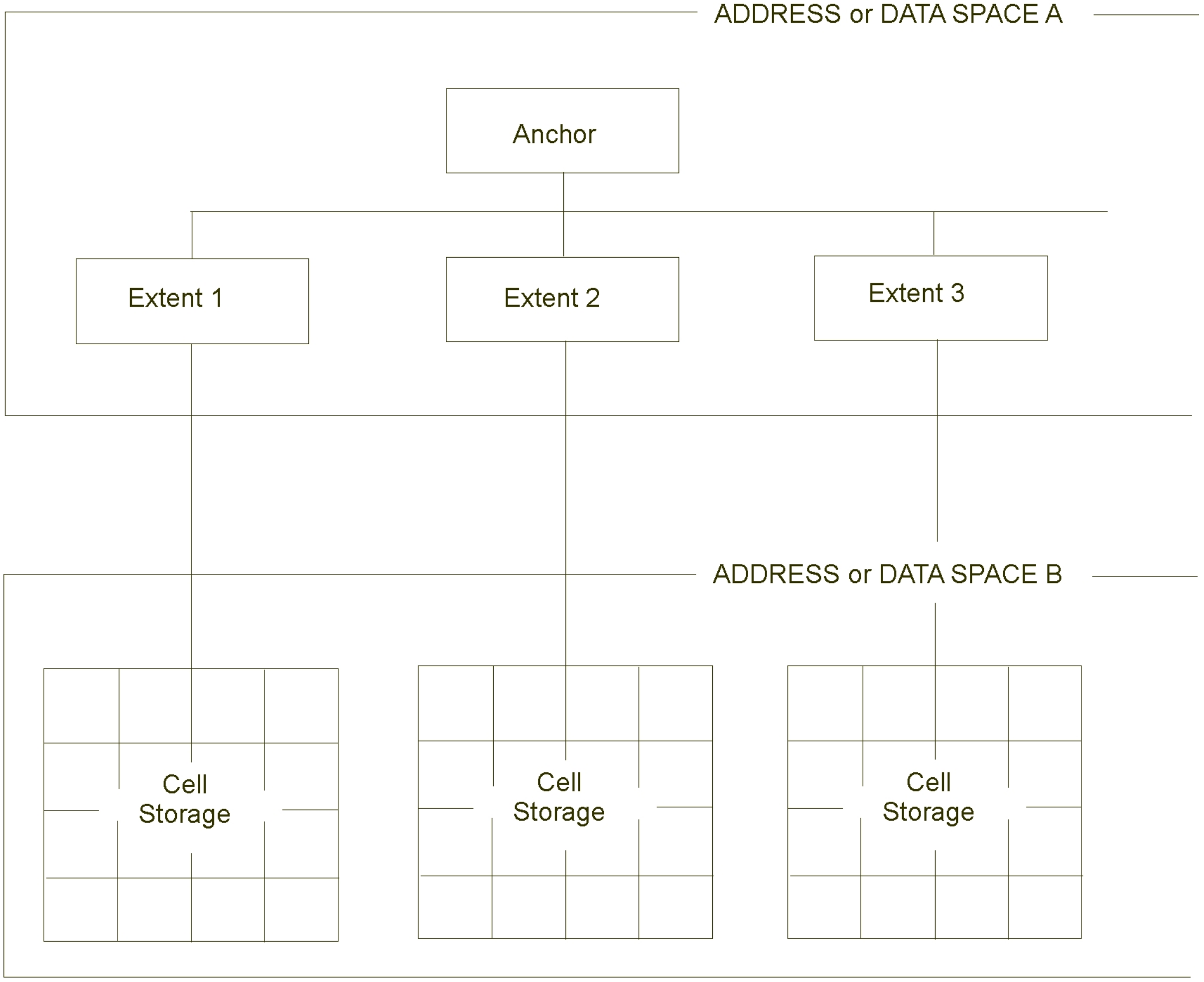


Figure 32. Cell Pool Storage

Storage Considerations

When you plan the size of the cell storage, consider the total requirements of your application for this storage and some performance factors. A single extent can control any number of cells up to 2^{24} (16,777,216) cells.

The size of a single extent can be up to 2^{24} - 1 bytes. You can have multiple extents for performance purposes.

Avoid having a large number of extents, where each extent is responsible for a small number of cells. In general, a greater requirement for cells should mean a proportionately smaller number of extents. The following two examples illustrate this point.

- If you have 10,000 cells in the pool, a good extent size is 2,500 cells per extent.
- If you have 100,000 cells in the pool, a good extent size is 10,000 cells per extent.

"Cell Pool Services Coding Example" on page 114 contains an example of using callable cell pool services with data spaces. It also describes some storage considerations.

Link-Editing Programs Using Callable Cell Pool Services

Any program that invokes cell pool services must be link-edited with an IBM-provided linkage-assist routine. The linkage-assist routine provides the logic needed to locate and invoke the callable services. The linkage-assist routine resides in IJSYSRS.SYSLIB. The following example shows the job stream required for link-editing a program with the linkage-assist routine.

```
// JOB LNKJOB
// LIBDEF PHASE,CATALOG=libray.sublibrary
// OPTION CATAL
  PHASE userprog,*
  INCLUDE userprog
  INCLUDE CSRCPOOL
/*
// EXEC LNKEDT
/&
```

The example assumes that the program you are link-editing is reentrant.

Using Callable Cell Pool Services

The following sections describe how you can use callable cell pool services to control storage and request information about the cell pools. The discussion of creating a cell pool and adding an extent assumes that you have already obtained the storage for these areas.

The CALL Macro

Cell pool services are invoked through the CALL macro.

The CALL macro passes control to another program at a specified entry point. The linkage established when control is passed is the same as when using the BAL instruction: the issuing program expects control to be returned.

The CALL macro was modified with VSE/ESA 2.1 and is compatible with the z/OS* macro of the same name.

Characteristics of the CALL Macro:

If a control section is not part of the object module which applies the CALL macro, the linkage editor attempts to resolve this external reference by including the object module which contains the control section (AUTOLINK feature). When the CALL macro is executed, control is passed to the control section at the specified entry point.

Further characteristics and restrictions:

- Programs in either the primary or the access register (AR) ASC mode can invoke the CALL macro.
- An address parameter list can be constructed and a calling sequence identifier can be provided.
- The CALL macro does not generate any return codes. A return code in GPR 15 or AR 15 originates from the called program.
- Control parameters must be in the caller's primary address space.
- The CALL macro cannot be used to pass control to a program in a different addressing mode (24-bit or 31-bit) than the calling program is.

Available Cell Pool Services

To use cell pool services, a program must issue the **CALL** macro specifying one of the following services:

- **CSRPBLD**: Create a cell pool and initialize an anchor.
- **CSRPEXP**: Expand a cell pool by adding an extent.
- **CSRPCON**: Connect cell storage to an extent.
- **CSRPACT**: Activate previously connected storage.
- **CSRPDAC**: Deactivate an extent.
- **CSRPDIS**: Disconnect the cell storage for an extent.
- **CSRPGET** or **CSRPRGT**: Allocate a cell from a cell pool.
- **CSRPFRE** or **CSRPRFR**: Return a cell to the cell pool.
- **CSRPQPL**: Query the cell pool.

- **CSRPQEX**: Query a cell pool extent.
- **CSRPQCL**: Query a cell.

Further Information: The syntax and parameter description of the CALL macro and the cell pool services are provided in z/VSE System Macros Reference. A coding example is provided in this documentation. Refer to "Cell Pool Services Coding Example" on page 114.

Creating a Cell Pool

To create a cell pool, call the CSRPBLD service. This service initializes the anchor for the cell pool, assigns the name of the pool, and establishes the size of the cells.

Adding an Extent and Connecting it to the Cell Storage

To add an extent and connect it to the cell storage, call the CSRPEXP service. You need at least one extent in a cell pool. Each extent is responsible for one cell storage area. You can add an extent to increase the numbers of cells; the maximum number of extents in a cell pool is 65,536. The CSRPEXP service initializes an extent for the cell pool, connects the cell storage area to the extent, and activates the cell storage for the extent.

Having activated the cell storage for an extent, you can now process CSRPGET requests from the cells that the extent represents.

Contracting a Cell Pool, Deactivating its Extents, and Disconnect its Storage

To contract a cell pool, deactivate its extents, and disconnect its storage, use the CSRPDAC and CSRPDIS services. CSRPDAC deactivates an extent and prevents the processing of any further CSRPGET requests from the storage that the extent represents. Cell FREE (CSRPFRE) requests are unaffected. You can use the CSRPACT service to reactivate an inactive extent (which was deactivated with CSRPDAC).

CSRPDIS disconnects the cell storage from an extent and makes cell storage unavailable. After you disconnect an extent, you can free the cell storage associated with the extent.

Reusing a Deactivated and Disconnected Extent

To reuse a deactivated and disconnected extent, call the CSRPCON and CSRPACT services, not CSRPEXP. This is generally the only time you will need to use these two services. CSRPCON reconnects an extent to cell pool storage that you have not explicitly freed or that connects the extent to cells in newly-obtained storage. If you reconnect the extent to new cell storage, be sure that the extent is large enough to support the size of the cell storage. (CSRPCON undoes the effects of using CSRPDIS.) CSRPACT activates the cell storage for the extent. You can now issue CSRPGET requests for the cells.

Allocating Cells and Deallocate Previously Allocated Cells

To allocate (that is, obtain) cells and deallocate (that is, free) previously allocated cells, you can choose between two forms of the same services. One service form supports the standard CALL interface. The other supports a register interface and is appropriate for programs that cannot obtain storage for a parameter list. The two service functions are identical; however, the calling interface is different.

The CSRPGET (standard CALL interface) and CSRPRGT (register interface) services allocate a cell from the cell pool. You can allocate cells only from extents that have not been deactivated. Such an extent is called an **active extent**. The services return to the caller the address of the allocated cell.

The CSRPFRE (standard CALL interface) and CSRPRFR (register interface) services return a previously allocated cell to a cell pool. They return a code to the caller if they cannot find the cell associated with an extent. If you free the last allocated cell in an inactive extent, you will receive a unique code. You may use this information to initiate cell pool contraction.

Obtaining Status Information About a Cell Pool

To obtain status information about a cell pool, use one of three services. These services do not prevent the cell pool from changing during a status query. They return status as it is at the time you issue the CALL.

The CSRPQPL service returns information about the entire cell pool. It returns the following:

- Pool name
- Cell size
- Total number of cells in active extents
- Total number of available cells associated with active extents
- Number of extents in the cell pool.

The CSRPQEX service returns information about a specific extent. It returns the following:

- Address and length of the extent
- Address and length of the cell storage area
- Total number of cells associated with the extent
- Number of available cells associated with the extent.

The CSRPQCL service returns information about a cell. It returns the following:

- Number of the extent that represents the cell
- Cell allocation status.

Invocation Requirements

The requirements for the calling program are:

Authorization
Problem state or supervisor state.

AMODE
24-bit or 31-bit addressing mode. All input addresses must be valid 31-bit addresses.

ASC mode
Primary or AR mode.

If the anchor and the extents are located in a data space, the caller must be in AR mode.

If an AR mode program invokes the macro, it must first issue SYSSTATE ASCENV=AR to ensure that the CALL macro generates the correct code for AR mode.

Control parameters
All parameters must reside in a single address or data space, and must be addressable by the caller. They must be in the primary address space or in an address/data space that is addressable through a public entry on the caller's dispatchable unit access list (DU-AL).

Register Usage

Some callers depend on register contents remaining the same before and after requesting a service. If the system changes the contents of registers on which the caller depends, the caller must save them before requesting the service, and restore them after the system returns control.

The exact register usage is shown for each cell pool service in z/VSE System Macros Reference.

Return Codes

Each time you call a cell pool service, you receive a return code. The return code indicates whether the service completed successfully, encountered an unusual condition, or was unable to complete successfully. When you receive a return code that indicates a problem or an unusual condition, your program can either attempt to correct the problem, or can terminate its processing.

The services pass return codes in both the parameter list and in register 15. Return codes are described for each cell pool service in z/VSE System Macros Reference.

Cell Pool Services Coding Example

This example shows how to use cell pool services. The anchor, the one extent, and the cell storage are all in the same data space. The caller obtains a cell from the cell storage area and requests information about the pool, the extent, and the cell.

```
         CSRCPASM              INVOKE CELL POOL SERVICES ASSEMBLER DECLARES
         SAC   512             SET AR ASC MODE
         SYSSTATE ASCENV=AR
*
* Establish addressability to code.                              *
*
         LAE   AR12,0
         BASR  R12,0
         USING *,R12
*
* Get data space for the cell pool.                              *
*
GETDSP   DSPSERV CREATE,NAME=DSPCNAME,STOKEN=DSPCSTKN,                 X
               BLOCKS=DSPBLCKS,ORIGIN=DSPCORG
*
* Add the data space to caller's access list.                    *
*
GETALET  ALESERV ADD,STOKEN=DSPCSTKN,ALET=DSPCALET,AL=WORKUNIT
         L     2,DSPCORG        ORIGIN OF SPACE IN GR2
         ST    2,DSPCMARK       DSPCMARK IS MARK FOR DATA SPACE
*
* Copy ALET to ANCHALET for calls to cell pool services.         *
*
         MVC   ANCHALET(4),DSPCALET
*
* Set address and size of the anchor
*
         L    R4,DSPCMARK
         ST   R4,ANCHADDR
         A    R4,ANCHSIZE
         ST   R4,DSPCMARK
*
* Call the build service.                                        *
*
         CALL CSRPBLD,(ANCHALET,ANCHADDR,USERNAME,CELLSIZE,RTNCODE)
*
* Set address and size of the extent and connect extent to cells *
*
         L    R4,DSPCMARK               RESERVES
         ST   R4,XTNTADDR
         A    R4,XTNTSIZE               SETS SIZE OF EXTENT
         ST   R4,CELLSTAD
         A    R4,CELLSTLN               SETS SIZE OF CELL STORAGE
         ST   R4,DSPCMARK               DATA
         CALL CSRPEXP,(ANCHALET,ANCHADDR,XTNTADDR,XTNTSIZE,            X
               CELLSTAD,CELLSTLN,EXTENT,RTNCODE)
** Get a cell.  CELLADDR receives the address of the cell.       *
*
         CALL CSRPGET,(ANCHALET,ANCHADDR,CELLADDR,RTNCODE)
*
* The program uses the cells.
*
* Query the pool, the extent, and the cell.                      *
*
         CALL CSRPQPL,(ANCHALET,ANCHADDR,QNAME,QCELLSZ,QTOT_CELLS,     X
               QAVAIL_CELLS,QNUMEXT,QRTNCODE)
         CALL CSRPQEX,(ANCHALET,ANCHADDR,EXTENT,QEXSTAT,QXTNT_ADDR,    X
               QXTNT_LEN,QCELL_ADDR,QCELL_LEN,QTOT_CELLS,              X
               QAVAIL_CELLS,QRTNCODE)
         CALL CSRPQCL,(ANCHALET,ANCHADDR,CELLADDR,QCLAVL,QCLEXT,       X
               QRTNCODE)
*
* Free the cell.                                                 *
*
         CALL CSRPFRE,(ANCHALET,ANCHADDR,CELLADDR,RTNCODE)
*
* Deactivate the extent.                                         *
*
         CALL CSRPDAC,(ANCHALET,ANCHADDR,EXTENT,RTNCODE)
*
* Disconnect the extent.                                         *
*
         CALL CSRPDIS,(ANCHALET,ANCHADDR,EXTENT,QCELL_ADDR,QCELL_LEN,  X
               QRTNCODE)
*
* Remove the data space from the access list.                    *
*
         ALESERV DELETE,ALET=DSPCALET
*
* Delete the data space.                                         *
*
         DSPSERV DELETE,STOKEN=DSPCSTKN
*
* Return to caller.
*
         SAC 0                          ENSURE IN PRIMARY MODE
         SYSSTATE ASCENV=P              RESET GLOBAL BIT
         BR ...                         RETURN TO CALLER
*
*****************************************************************
* Constants and data areas used by cell pool services           *
*****************************************************************
*
CELLS_PER_EXTENT  EQU  512
EXTENTS_PER_POOL  EQU  10
CELLSIZE_EQU      EQU  256
CELLS_PER_POOL    EQU  CELLS_PER_EXTENT*EXTENTS_PER_POOL
XTNTSIZE_EQU      EQU  128+(((CELLS_PER_EXTENT+63)/64)*8)
STORSIZE_EQU      EQU  CELLS_PER_EXTENT*CELLSIZE_EQU
CELLS_IN_POOL     DC   A(CELLS_PER_POOL)ANCHALET  DS   F
ANCHADDR  DS   F
CELLSIZE  DC   A(CELLSIZE_EQU)
USERNAME  DC   CL8'MYCELLPL'
ANCHSIZE  DC   F'64'
XTNTSIZE  DC   A(XTNTSIZE_EQU)
XTNTADDR  DS   F
CELLSTAD  DS   F
CELLSTLN  DC   A(STORSIZE_EQU)
CELLADDR  DS   F
EXTENT    DS   F
STATUS    DS   F
RTNCODE   DS   F
*
*****************************************************************
* Constant data and areas for data space                        *
*****************************************************************
*
         DS    0D
DSPCSTKN DS    CL8                      DATA SPACE STOKEN
DSPCORG  DS    F                        DATA SPACE ORIGIN RETURNED
DSPCSIZE EQU   STORSIZE_EQU*EXTENTS_PER_POOL 1.28MB DATA SPACE
DSPBLCKS DC    A((DSPCSIZE+4095)/4096)  BLOCKS FOR 1.28MB DATA SPACE
DSPCALET DS    F
DSPCMARK DS    F                        HIGH WATER MARK FOR DATA SPACE
DSPCNAME DC    CL8'DATASPC1'            DATA SPACE NAME
*
*****************************************************************
* Values returned by queries                                    *
*****************************************************************
*
```

```
QNAME          DS   CL8
QCELLSZ        DS   F
QNUMEXT        DS   F
QEXTNUM        DS   F
QEXSTAT        DS   F
QXTNT_ADDR     DS   F
QXTNT_LEN      DS   F
QCELL_ADDR     DS   F
QCELL_LEN      DS   F
QTOT_CELLS     DS   F
QAVAIL_CELLS   DS   F
QRTNCODE       DS   F
RC             DS   F
QCLADDR        DS   F
QCLEXT         DS   F
QCLAVL         DS   F
```

Appendix A. Linkage Editor and Librarian Support

Linkage Editor Support for 31-Bit Addressing

The linkage editor assigns an AMODE (addressing mode) and a RMODE (residency mode) value for a phase as defined in one of the following:

- The AMODE and RMODE flags in the ESD (external symbol dictionary) data (set, for example, by the High Level Assembler).
- The AMODE and RMODE values provided as a parameter in the PARM field of the EXEC LNKEDT statement.
- The AMODE and RMODE operands provided via the linkage editor MODE control statement.

The linkage editor stores the AMODE and RMODE values assigned to a phase into the directory entry of that phase.

To support portability among different systems, the z/VSE linkage editor determines AMODE and RMODE values in a similar way as does the linkage editor of z/OS.

Maximum Size of a Phase

The maximum size of a phase that can be link-edited is:

```
16 MB  minus  "specified origin in the PHASE card"
```

For example, if "S" is specified as origin, the maximum size is 16 MB minus the partition start address (of the partition in which the linkage editor is running).

Assigning the AMODE

The addressing mode (AMODE) is the attribute of the entry point of the loaded phase. It specifies the addressing mode that will be in effect when the phase is entered at that entry point at execution time.

The linkage editor will assign the addressing mode for the entry point according to the following rules:

1. A default AMODE of 24 is assigned.
2. If the AMODE is specified in the ESD data for the entry point (set by the High Level Assembler, for example), the linkage editor assigns that AMODE to the phase.
3. If the AMODE is specified as a parameter in the PARM field of the EXEC LNKEDT statement, the linkage editor assigns that AMODE to the phase. This AMODE value overrides the AMODE value, if any, found in the ESD data.
4. If the AMODE is specified as an operand in the MODE control statement, the linkage editor assigns that AMODE to the entry point of the phase. This AMODE value overrides any AMODE value specified as an operand in the PARM field of the EXEC LNKEDT statement or any AMODE values found in the ESD data.

Additional Considerations:

The AMODE value provided in the ESD data of the object modules for the entry point of the phase is retained in the directory entry of the phase. That is, when the phase is punched with the VSE librarian the ESD card for the phase in the punched deck contains the AMODE value assigned from the ESD data.

In addition, if the AMODE was assigned during link editing from the overriding AMODE specifications (from the PARM field of the EXEC statement or the MODE control statement), a MODE control statement is punched following the PHASE card in the punched deck containing this overriding AMODE value.

Assigning the RMODE

The residency mode (RMODE) is the attribute of the phase that specifies where the phase can reside in virtual storage.

The linkage editor determines the residency mode for a phase according to the following rules:

1. A default RMODE of 24 is assigned.
2. If the RMODE is specified in the ESD data (set by the H Assembler, for example) the linkage editor assigns the RMODE value from the control section or private code that contributes to the phase.

   As the control sections and private code that contribute to the phase are processed, the RMODE value for the phase, based on the ESD data, is accumulated on a "most restrictive" basis. This means:

   - If any section in the phase has an RMODE of 24, the RMODE for the phase is 24.
   - If all sections in the phase have an RMODE of ANY, the RMODE for the phase is ANY.
3. If the RMODE is specified as a parameter in the PARM field of the EXEC LNKEDT statement the linkage editor assigns that RMODE to the phase. This RMODE value overrides the RMODE value, if any, found in the ESD data.
4. If the RMODE is specified as an operand in the MODE control statement the linkage editor assigns that RMODE to the phase. This RMODE value overrides any RMODE value specified as an operand in the PARM field of the EXEC LNKEDT statement or any RMODE values found in the ESD data.

Additional Considerations:

- If PHASE statements are encountered during link editing which are needed to construct Overlay Programs (that is, where the origin in the PHASE statement is **ROOT** or a **symbol**, or the origin is a ***** and it is not the first phase), all phases linked in this link edit job step are assigned an RMODE of 24, regardless of the ESD data, the PARM field parameter, or the MODE control statement operand.
- For non-relocatable phases, RMODE=ANY is not accepted.
- The RMODE value provided in the ESD data of the object modules for the entry point of the phase is retained in the directory entry of the phase (except for the case that PHASE statements are encountered during link editing which are needed to construct Overlay Programs); that is, when the phase is punched with the VSE librarian, the ESD card for the phase in the punched deck contains the RMODE value assigned from the ESD data.

  In addition, if the RMODE was assigned during link editing from the overriding RMODE specifications (from the PARM field of the EXEC statement or the MODE control statement), a MODE control statement is punched following the PHASE card in the punched deck containing this overriding RMODE value.

AMODE/RMODE Hierarchy

The following hierarchy is used to determine the addressing and residency modes assigned to a phase:

1. AMODE/RMODE values of the linkage editor MODE statement. The AMODE/RMODE values from the MODE statement override the AMODE/RMODE values from the PARM field of the EXEC LNKEDT statement and the ESD data.
2. AMODE/RMODE values of the PARM field of the EXEC LNKEDT statement.

   The AMODE/RMODE values of the PARM field of the EXEC LNKEDT statement override the AMODE/RMODE values from the ESD data.
3. AMODE/RMODE values of the ESD data produced by the AMODE or RMODE assembler statement of the High Level Assembler.
4. A default value of 24.

Additional Considerations:

- The specification of the AMODE/RMODE values in the MODE control statement applies **only** to the phase which includes the MODE control statement.

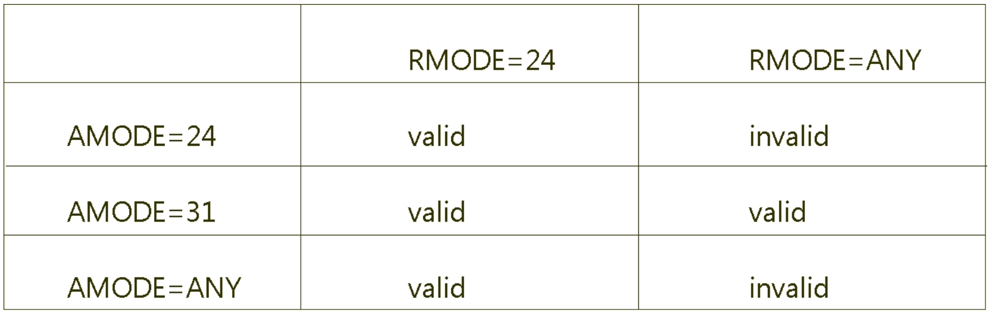
The specification of the AMODE/RMODE values in the PARM field of the EXEC LNKEDT statement applies to **all** phases listed in the link-edit job step unless they are overridden for a specific phase by a MODE control statement.

- A phase produced from multiple object modules results in an RMODE of 24, if any one of the object modules has an RMODE of 24 (unless it is overridden by the linkage editor MODE control statement or the PARM field of the EXEC LNKEDT statement).
- The use of PHASE statements where the origin for the phase is specified as **ROOT** or as **symbol**, or as ***** and it is not the first phase in the link-edit job step, results in an RMODE of 24 for all phases linked in that link-edit job step.

  On the other hand, you get an RMODE of ANY with PHASE statements where the origin for the phase is specified as **S** or ***** and it is the first phase in the link-edit job or a displacement must be used. This means, in principle an RMODE of ANY is assigned only to a single linked phase or to re-linked phases (which have been punched by the librarian, for example). The librarian inserts an **S** as the origin into the punched PHASE statement for relocatable phases.

AMODE/RMODE Combinations in the MODE Control Statement

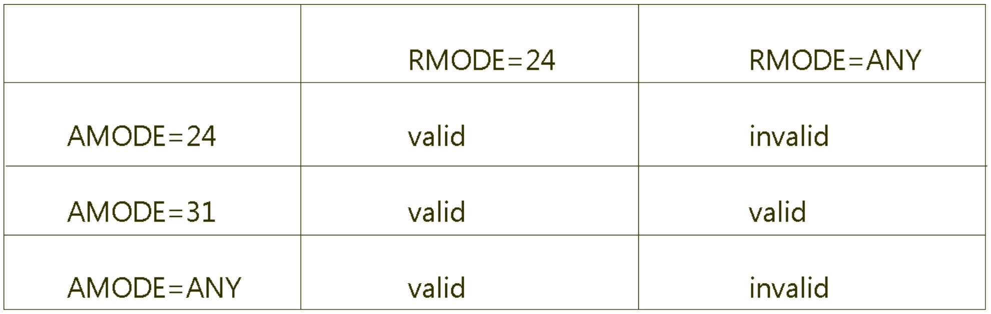
The linkage editor validates the combination of the AMODE value and the RMODE value, as specified by the user in the MODE control statement, according to the following table:



	RMODE=24	RMODE=ANY
AMODE=24	valid	invalid
AMODE=31	valid	valid
AMODE=ANY	valid	invalid

AMODE/RMODE Combinations in the PARM Field

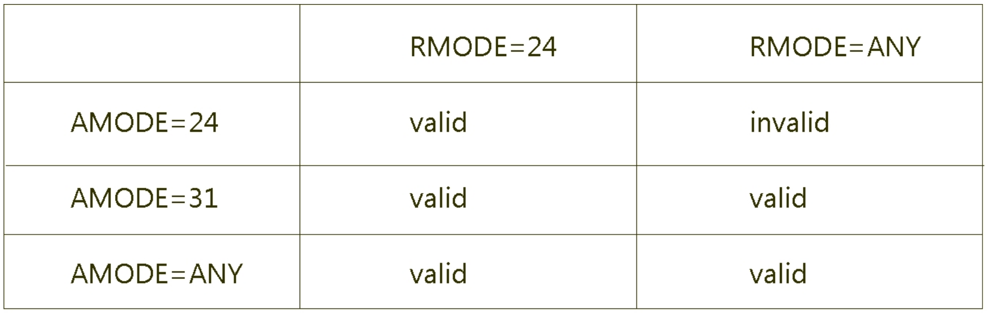
The linkage editor validates the combination of the AMODE value and the RMODE value, as specified by the user in the PARM field of the EXEC LNKEDT statement, according to the following table:



	RMODE=24	RMODE=ANY
AMODE=24	valid	invalid
AMODE=31	valid	valid
AMODE=ANY	valid	invalid

AMODE/RMODE Combinations from the ESD

When AMODE and RMODE data have not been specified on either the linkage editor MODE control statement or in the PARM field of the EXEC LNKEDT statement, the linkage editor determines the AMODE for the entry point of the phase and the RMODE for the phase based on the ESD data. The linkage editor validates the AMODE/RMODE combinations from the ESD as follows:



	RMODE=24	RMODE=ANY
AMODE=24	valid	invalid
AMODE=31	valid	valid
AMODE=ANY	valid	valid

Note: An AMODE/RMODE combination ANY/ANY from the ESD data is valid (contrary to the MODE control statement and the PARM field). The reason is that the final AMODE/RMODE combination for a phase can be determined from the CSECTs of a phase as described below.

The entry point of a phase may be specified either by the external symbol of a control section or an entry name. When an entry point is a control section name, the linkage editor acquires the AMODE and RMODE directly from the control section name ESD entry. When an entry point is an entry name external symbol, the linkage editor acquires the AMODE and RMODE data from the associated control section name ESD entry.

Based on the AMODE/RMODE data acquired from the ESD, the linkage editor determines the RMODE of the phase, and assigns an AMODE to the entry point of the phase as follows:

- If the external symbol of the entry point is marked with any of the allowed AMODE values and an RMODE of 24, the entry point of the phase is assigned the same AMODE attribute as its associated external symbol.
- The AMODE 24/RMODE ANY combination is invalid as it could allow 24-bit addressing above the 16 MB line. If the linkage editor does find this combination, it issues a warning message on SYSLST, forces the RMODE of the phase to 24, and assigns an AMODE of 24 to the entry point of the phase.
- If the external symbol of the entry point is marked with AMODE 31/RMODE ANY, the entry point of the phase is assigned an AMODE 31 and the RMODE will be that of the phase; (which is the RMODE accumulated on the "most restrictive" basis as described under “Linkage Editor RMODE Processing” on page 18).
- If the external symbol of the entry point is marked with AMODE ANY/RMODE ANY, the entry point of the phase is assigned an AMODE and RMODE according to the following hierarchy:
  - If the phase contains one or more CSECTs marked AMODE 24, the linkage editor assigns an AMODE of 24 to the entry point of the phase.
  - If the phase has an RMODE of 24 and it contains no CSECTs marked AMODE 24, the linkage editor assigns an AMODE of ANY to the entry point of the phase.
  - If the RMODE of the phase is ANY, the linkage editor assigns an AMODE of 31 to the entry point of the phase.

Handling of Invalid AMODE/RMODE Combinations

If the linkage editor finds the invalid AMODE/RMODE combination AMODE(24)/RMODE(ANY) in the input ESD card, it issues warning message 2174I on SYSLST and forces the RMODE of the CSECT to 24.

If the AMODE/RMODE combination resulting from the EXEC LNKEDT statement is invalid, the linkage editor issues the warning message 2175I on SYSLST and ignores the AMODE/RMODE values from the PARM field.

If the AMODE/RMODE combination resulting from the linkage editor MODE control statement is invalid, the linkage editor issues the warning message 2176I on SYSLST and ignores the AMODE/RMODE operands from the MODE control statement.

Further Information

For a detailed description of linkage editor statements such as MODE and EXEC LNKEDT and how to use them, refer to z/VSE System Control Statements under "Linkage Editor".

Notes on the MODE Control Statement

To assign the addressing mode (AMODE) for the entry point of a phase and the residency mode (RMODE) for a phase, the MODE control statement has been introduced for the linkage editor.

The MODE control statement must follow the PHASE card of a phase. If more than one MODE control statement is encountered during the link-editing of a phase, the mode specification from the first valid MODE control statement is used.

AMODE and RMODE specifications are not handled independently, that is, if only one value, either AMODE or RMODE, is specified in the MODE control statement, the other value is implied according to the values shown in Table 6 on page 121.

Notes on the EXEC LNKEDT Statement

To assign the addressing mode for the entry point of a phase, the AMODE/RMODE parameters have been introduced in the PARM field of the EXEC LNKEDT statement.

The AMODE/RMODE values specified in the PARM field are valid for **all** phases linked in a link-editing job except for those phases for which a MODE control statement has been specified.

AMODE and RMODE are not handled independently, that is, if only one value, either AMODE or RMODE, is specified in the PARM field, the other value is implied according to the values shown in Table 6 on page 121.

Table 6. Implied AMODE or RMODE

Value specified	Value implied
AMODE = 24	RMODE = 24
AMODE = 31	RMODE = 24
AMODE = ANY	RMODE = 24
RMODE = 24	(see Note below)
RMODE = ANY	AMODE = 31

Note: If only an RMODE value of 24 is specified in the MODE control statement or in the PARM field of the EXEC LNKEDT statements, no overriding AMODE value is assigned. Instead, the AMODE value in the ESD data for the entry point is used.

Librarian Support for 31-Bit Addressing

The new program attributes AMODE and RMODE are stored in the directory entry of a phase and in the ESD card for punched OBJ members. To support AMODE and RMODE, the following librarian functions have been extended:

1. Librarian PUNCH
2. Librarian LISTDIR
3. SET SDL Processing

Punching a Phase

The librarian PUNCH (punch member contents) command stores in the ESD card of the punched object the AMODE and RMODE as originally specified in the ESD data. In addition, if the AMODE or RMODE was assigned during link editing from the overriding specifications (from the linkage editor PARM field or MODE control statement) a MODE control statement is punched following the PHASE card.

Thus, you can get the original AMODE/RMODE specifications from the ESD data of the program by relinking the phase without the generated MODE control statement.

LISTDIR Output

The layout of LISTDIR (list directory information) output has been changed. It now shows the new attributes AMODE and RMODE for each phase stored in a library/sublibrary. The documentation z/VSE Guide to System Functions provides further details about the librarian LISTDIR command under "List Library, Sublibrary, or Member Information".

SET SDL Processing

Starting with VSE/ESA 1.3, a second SVA (shared virtual area), here referred to as SVA (31-Bit), has been introduced. This SVA resides at the high end of the address space which is usually above 16 MB and SVA phases with RMODE ANY are loaded into the VLA (virtual library area) of this area.

The new "high" VLA is an extension of the "old" VLA in the SVA (24-Bit) area. As with previous releases, there is only one SDL (system directory list) located in the SVA (24-Bit) but this SDL also addresses the phases in the high VLA.

The SET SDL function tries to load RMODE ANY phases to the high VLA first. If the space is not sufficient, the VLA in the SVA (24-Bit) is selected.

Appendix B. Macro and Command Support

z/VSE Macros and Their Mode Dependencies

Table 7 on page 123 lists the z/VSE macros and the modes allowed at execution time. The list mainly applies to the 31-bit addressing support, except for the AR MODE column which applies to the data space support, and the AMODE 64 column which applies to the 64-bit virtual support.

- AMODE indicates the *addressing mode* that is expected to be in effect when the program is entered. AMODE can have one of the following values:

  AMODE 24
  Specifies 24-bit addressing mode.

  AMODE 31
  Specifies 31-bit addressing mode.

  AMODE 64
  Specifies 64-bit addressing mode.

  An **X** in column AMODE 24, AMODE 31, or AMODE 64 indicates that the macro can be invoked by a program executing in that mode. An **X** in both AMODE 24 and AMODE 31 columns implies AMODE ANY which indicates that the macro can be used and executed in 24-bit and 31-bit addressing mode. AMODE ANY does *not* imply 64-bit addressing mode.

- RMODE indicates the *residency mode*, that is, the virtual storage location where the program must reside. RMODE can have one of the following values:

  RMODE 24
  Indicates that the program must reside in virtual storage below 16 MB.

  RMODE ANY
  Indicates that the program can reside anywhere in virtual storage, either above or below 16 MB, but always *below the 2 GB bar*. RMODE ANY does *not* imply storage above the bar.

  An **X** in column RMODE 24 or RMODE ANY indicates that the macro can be invoked by a program executing in that mode.

  The parameter list of a requested macro must have the same RMODE as the macro itself. The most important exceptions are pointed out in the 'Comments' column; for more details, see the corresponding macro description.

- AR MODE: Most macros listed in Table 7 on page 123 can only be called in *primary* ASC (address space control) mode, except those that have an indication for *Access Register (AR)* ASC mode in column **AR MODE** (for macros supporting data spaces). An **X** in that column indicates that both primary and AR mode are possible.

Most macro services based on branch and link interfaces do **not** check for execution mode violations, that is, the program requesting the macro service is responsible for the correct execution mode (AMODE and RMODE). An execution mode violation may lead to unpredictable results.

Further Information: The following table reflects the z/VSE macro support as documented in the manual z/VSE System Macros Reference. For VSE/VSAM and VSE/POWER macros and their mode dependencies, refer to the manuals VSE/VSAM Commands and VSE/POWER Application Programming.

Table 7. z/VSE macros and their mode dependencies

Macro Name	AMODE			RMODE		AR MODE	Comments
	24	31	64	24	ANY		
ALESERVxx	x	x		x	x	x	

Table 7. z/VSE macros and their mode dependencies (continued)

Macro Name	**AMODE**			**RMODE**		**AR MODE**	**Comments**
	24	**31**	**64**	**24**	**ANY**		
AMODESWxx	x	x		x	x		
ASPL				x	x		
ASSIGN	x	x		x	x		
ATTACH	x	x		x	x		Subtask save area RMODE=24
AVRLIST				x			
CALL	x	x		x	x	x	
CALL CSxx	x	x		x	x	x	Caller must be in AR mode if anchor and extents are located in a data space
CANCEL	x	x		x	x		
CCB				x	x		VSE/POWER supports RMODE=24 only
CDDELETE	x	x		x	x		
CDLOAD	x	x		x	x		
CDMOD	x			x			
CHAP	x	x		x	x		
CHECK	x			x			
CHKPT	x			x			
CLOSE|CLOSER	x	x		x			DTF has to be allocated below 16MB
CNTRL	x			x			
COMRG	x	x		x	x		
CPCLOSE	x			x			
CSRCMPSC		x				x	
CSRYCMPS				x			
DCTENTRY				x			
DEQ	x	x		x	x		
DETACH	x	x		x	x		Subtask save area RMODE=24
DIMOD	x			x			
DOM	x	x		x			
DSPSERVxx		x		x	x	x	AR mode: SYSSTATE required
DTFxx				x			

Table 7. z/VSE macros and their mode dependencies (continued)

Macro Name	AMODE			RMODE		AR MODE	Comments
	24	31	64	24	ANY		
DTL	x	x		x	x		
DUMODFx	x			x			
DUMP	x	x		x			
ENDFL	x			x			
ENQ	x	x		x	x		
EOJ	x	x		x	x		RMODE=ANY if RC is omitted
ERET	x			x			
ESETL	x			x			
EXCP	x	x		x	x		Control blocks RMODE=24
EXIT	x	x		x	x		
EXTRACT	x	x		x	x		
FCEPGOUT	x			x			SPLEVEL SET=1
	x	x		x	x		SPLEVEL SET>1
FEOV	x			x			
FEOVD	x			x			
FETCH	x	x		x			
FREE	x			x			
FREEVIS	x	x		x	x		
GENDTL	x	x		x			
GENIORB	x			x			
GENL	x	x		x			
GET	x			x			
GETIME	x	x		x	x		
GETSYMB	x	x		x	x		
GETVCE	x	x			x		
GETVIS	x	x		x	x		
IARV64		x	x		x	x	
IJBPUB	N/A	N/A	N/A	N/A	N/A		
IJJLBSER				x	x		
IORB				x			
ISMOD	x			x			
JDUMP	x	x		x			
JOBCOM	x			x			

Table 7. z/VSE macros and their mode dependencies (continued)

Macro Name	**AMODE**			**RMODE**		**AR MODE**	**Comments**
	24	**31**	**64**	**24**	**ANY**		
LBRET	x			x			
LBSERV xx	x	x			x		
LFCB	x			x			
LIBRDCB	x	x			x		
LIBRM xxx	x	x			x		
LOAD	x	x		x	x		RMODE=24 when LIST, SYS, DE, TXT, MFG, or RET specified; RMODE=ANY not allowed with parameter list
LOCK	x	x			x		
MAPDBY				x	x		
MAPBDYVR				x	x		
MAPDNTRY							
MAPEXTR				x	x		
MAPSAVAR				x	x		
MAPSSID	N/A	N/A		N/A	N/A		
MAPSYSP	N/A	N/A		N/A	N/A		
MAPXPCCB	N/A	N/A		N/A	N/A		
MODDTL	x	x		x			
MVCOM	x			x			
NOTE	x			x			
OPEN|OPENR	x	x		x			DTF has to be allocated below 16MB
PAGEIN	x			x			SPLEVEL SET=1
	x	x		x	x		SPLEVEL SET>1
PDUMP	x	x		x			
PFIX	x			x			SPLEVEL SET=1
	x	x		x	x		SPLEVEL SET>1
PFREE	x			x			SPLEVEL SET=1
	x	x		x	x		SPLEVEL SET>1
POINTR	x			x			
POINTS	x			x			
POINTW	x			x			
POST	x	x		x	x		

Table 7. z/VSE macros and their mode dependencies (continued)

Macro Name	**AMODE**			**RMODE**		**AR MODE**	**Comments**
	24	**31**	**64**	**24**	**ANY**		
PRMOD	x			x			
PRTOV	x			x			
PUT	x			x			
PUTR	x			x			
QSETPRT	x			x			
RCB				x	x		
READ	x			x			
REALAD	x	x		x	x		
RELEASE	x			x			
RELPAG	x			x			SPLEVEL SET=1
	x	x		x	x		SPLEVEL SET>1
RELSE	x			x			
RETURN	x	x		x	x		Returns with mode of issuer
RUNMODE	x			x			
SAVE	x	x		x	x		
SDUMP|SDUMPX	x	x		x	x	x	AR mode: SYSSTATE required
SECTVAL	x	x		x	x		
SEOV	x			x			
SETFL	x			x			
SETIME	x	x		x	x		
SETL	x			x			
SETPFA	x			x			
SETPRT	x			x			
SPLEVEL	N/A	N/A		N/A	N/A		
STXIT	x	x	x	x	x		
SUBSID	x	x		x	x		
SYSSTATE	x	x	x	x	x	x	
TECB				x	x		
TPIN	x			x			
TPOUT	x			x			
TRUNC	x			x			
TTIMER	x	x		x	x		

Table 7. z/VSE macros and their mode dependencies (continued)

Macro Name	AMODE			RMODE		AR MODE	Comments
	24	31	64	24	ANY		
UNLOCK	x	x			x		
VIRTAD	x	x		x	x		BTAM: AMODE=24 only
WAIT	x	x		x	x		
WAITF	x			x			
WAITM	x	x		x	x		
WRITE	x			x			
WTO	x	x		x	x		
WTOR	x	x		x	x		
XECBTAB	x			x			
XPCC	x	x		x	x		
XPCCB				x	x		
XPOST	x			x			
XWAIT	x			x			
YEAR224	x	x			x		

1. IPv6 not supported.
2. Supported, but with differences. For details refer to the IPv6/VSE Programming Guide.

Macro Support for 31-Bit Addressing

This section introduces selected macros that are new or that have been enhanced for the 31-bit addressing support. For the syntax and a detailed keyword and parameter description refer to the manual z/VSE System Macros Reference.

AMODESW Macro

The AMODESW macro can be used to:

- Switch the AMODE and, optionally, save a program's current AMODE.
- Switch the AMODE as part of a subroutine call and return.

For calling a subroutine and returning from it, AMODESW generates code reflecting the support provided by the instructions BASSM and BSM.

The AMODESW macro provides the following set of functions:

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

While the AMODESW macro allows a program to switch addressing modes, the user must make sure that the programs follow 24-bit or 31-bit addressing conventions.

Notes on Using the AMODESW Macro

1. Users must restore their program's addressability (set up the proper base register) on return from the call. You can use the address VSE returns in the return register to set up program addressability.
2. AMODESW CALL and AMODESW RETURN allow you to call and return from subroutines. You can use them anywhere you can use a BALR and BR sequence.

AMODESW Example

```
          .
          .
          AMODESW CALL,ADDRESS=MYSUB,AMODE=31,WR=(1)
          .
          .
          .
MYSUB    EQU  *
          .
          .
          .
          AMODESW RETURN
          .
          .
```

The sequence of instructions shown in the example does the following:

1. Calls via AMODESW the subroutine at label MYSUB by using a BASSM instruction.
2. Switches to 31-bit addressing mode.
3. Saves the return address and the addressing mode in register 14 (default value).
4. Returns via AMODESW to the caller and restores the addressing mode saved in register 14.

You can use the REGS and REG parameters on CALL and RETURN to override the registers used for the BASSM linkage.

Notes on Using AMODESW SET

1. AMODESW SET switches a program's addressing mode without requiring a branch to a subroutine. For example, to switch the current addressing mode to 31-bit addressing, a program might use:

   ```
   AMODESW SET,AMODE=31
   ```

2. To switch to a new mode from an unknown addressing mode and save the unknown mode, use the SAVE parameter. For example, the macro instruction

   ```
   AMODESW SET,AMODE=31,SAVE=(2)
   ```

   switches a program to 31-bit addressing mode and saves the current addressing mode as bit 0 of register 2. Only bit 0 of the SAVE register is altered. You can then use the value set by the SAVE parameter to restore the original addressing mode:

   ```
   AMODESW SET,AMODE=(2)
   ```

Storage Management Macros

GETVIS Macro

In order to access the partition GETVIS area above 16 MB or the system GETVIS area (31-Bit), a new GETVIS parameter has been introduced: **LOC**. It specifies the location of the virtual storage obtained by a GETVIS request. For LOC you can specify BELOW, ANY, or RES where RES means that the virtual storage is to be allocated depending on the callers location.

FREEVIS Macro

The FREEVIS macro is used to release virtual storage that was obtained by the GETVIS macro.

Page Management Macros

The macros FCEPGOUT, PAGEIN, PFIX, PFREE and RELPAG support 31-bit addresses. The address of the provided parameter list and the addresses in this list are treated as 3-byte addresses if the service is invoked in 24-bit mode and as 4-byte addresses if invoked in 31-bit mode. The end of the parameter list is indicated in 24-bit mode by any non-zero value and in 31-bit mode by bit 0 on in the byte following the last entry. Page management macros assembled by releases prior to VSE/ESA 1.3.0 or with SPLEVEL=1 must be executed in AMODE=24 and RMODE=24, otherwise the issuer is canceled.

By applying the SPLEVEL macro it is possible to generate downward compatible macro expansions.

PFIX Macro

The macro PFIX has been extended with the parameter RLOC to support PFIX of pages in real storage above 16 MB.

Program Load and Retrieval Macros

CDDELETE Macro

The CDDELETE macro deletes a phase previously loaded by a CDLOAD request into the partition GETVIS area.

Delete means that the phase load count is decreased by one. If the load count is zero the GETVIS storage occupied by the phase will be freed.

CDLOAD Macro

The CDLOAD Macro loads a phase into the **partition GETVIS area**. CDLOAD places the phase in virtual storage either below 16 MB or anywhere as indicated by the phase's RMODE. It gives control back to the caller. A loaded phase may cross the 16 MB line.

Note: The CDLOAD macro is recommended for loading phases, especially for loading phases above the 16 MB line. Together with the CDDELETE macro it provides enhanced GETVIS storage management.

FETCH Macro

The macro loads and gives control to the phase specified in the first operand (not back to the caller). FETCH can only be called below the 16 MB line (both expanded code and parameter lists must be below 16 MB).

GENL Macro

The macro generates a local directory list within the partition. It is required that both the local directory list and the macro expansion are located below 16 MB.

LOAD Macro

The LOAD macro loads a phase at the load point provided and returns control to the calling program. In case no load-point is provided by the user, the load-point specified at link-edit time (relocated) is used. Since the control is passed to the caller, the addressing mode is not changed by the load processing.

Note: The LOAD macro can only be used below 16 MB.

Task Communication Macros

ATTACH Macro

A subtask can be initiated by any other task of the partition with the ATTACH (attach a task) macro. ATTACH supports the **31-bit** as well as the **data space** environment. It can attach a subtask in 24-bit or 31-bit addressing mode (AMODE ANY) physically resident above or below 16 MB (RMODE ANY).

ATTACH issued in AMODE 24

The passed parameters are treated as 3-byte addresses.

ATTACH issued in AMODE 31

The passed parameters are treated as 4-byte addresses.

Table 8. ATTACH Macro and Its AMODE/RMODE Characteristics

Macro	Parameter	AMODE	RMODE	Comment
ATTACH		31	ANY	
	entrypoint	31	ANY	Entry point receives control in AMODE 31
	ABSAVE	-	ANY	
	ECB	-	ANY	
	MFG	-	ANY	
	NAME	-	ANY	
	SAVE	-	24	The subtask save area has to be allocated in RMODE 24

The attached task will get control in the same addressing mode the ATTACH issuer has. For example, if a main task issues the ATTACH macro in AMODE 31, the subtask will also receive control in AMODE 31.

The user supplied save area specified with the ABSAVE parameter of the **ATTACH** macro may reside below or above the 16 MB line. The old exit save area layout can only be used in a 24-bit environment. The new layout is extended by the saved access registers and the actual PSW.

DETACH Macro

The DETACH (detach task) macro terminates the execution of a task. A subtask is normally terminated by issuing a DETACH macro in the main task or in the subtask itself.

Note: The task's save area is always located below 16 MB (RMODE 24).

ENQ/DEQ Macro

A task protects or releases a resource by issuing an ENQ or DEQ macro. The ECB address in the RCB is treated as a 31-bit address.

ENQ/DEQ issued in AMODE 24

The RCB address is treated as a 24-bit address.

ENQ/DEQ issued in AMODE 31

The RCB address is treated as a 31-bit address.

POST Macro

The POST (post event) macro provides communication between two tasks in the same partition by posting an event control block (ECB). POST processing can post an ECB in 24-bit or 31-bit addressing mode (AMODE ANY) physically resident above or below 16 MB (RMODE ANY).

POST issued in AMODE 24

The passed parameters are treated as 24-bit addresses. All addresses (ECB, save area) are 24-bit addresses.

POST issued in AMODE 31

The passed parameters are treated as 31-bit addresses. All addresses (ECB, save area) are 31-bit addresses.

Table 9. POST Macro and Its AMODE/RMODE Characteristics

Macro	Parameter	AMODE	RMODE	Comment
POST		31	ANY	
	ECB	-	ANY	
	SAVE	-	24	The task save area has to be allocated in RMODE 24

WAIT Macro

With the WAIT (wait for event) macro, a task sets itself into the wait state until the event control block (ECB) specified in the macro is posted.

WAIT processing can wait for an ECB in 24-bit or 31-bit addressing mode (AMODE ANY) physically resident above or below 16 MB (RMODE ANY).

WAIT issued in AMODE 24

The ECB address is treated as a 24-bit address.

WAIT issued in AMODE 31

The ECB address is treated as a 31-bit address.

WAITM Macro

The WAITM macro works basically in the same way and has the same requirements as the WAIT macro except that it can handle multiple events.

WAITM issued in AMODE 24

The ECB addresses and the list address are treated as 24-bit addresses. The first byte following the last address in the list must be non-zero to indicate the end of the list.

WAITM issued in AMODE 31

The ECB addresses and the list address are treated as 31-bit addresses. The first bit of the last address in the list must be non-zero to indicate the end of the list.

I/O Processing Support for 31-Bit Addressing

CCB Macro

The CCB (channel control block definition) macro includes the keyword **CCW** to allow requests for the format-1 CCW (channel command word) as well as for the format-0 CCW.

- When I/O buffers are located above 16 MB, format-1 CCWs must be used to address these areas.
- VSE/VSAM uses format-1 CCWs and can therefore access I/O areas above 16 MB.

The following figures show the layout of the two formats:

CCW Formats

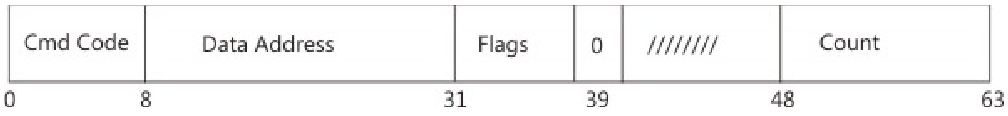


Figure 33. Format-0 CCW

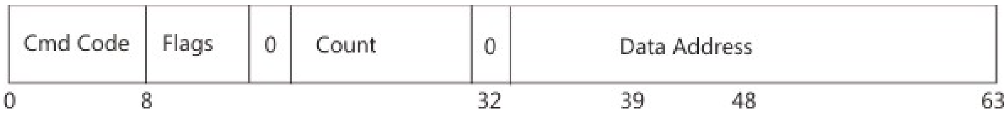


Figure 34. Format-1 CCW

Note: Use format-1 CCWs only if necessary. When running under VM, the format-1 CCW translation affects the VM guest performance.

Restrictions

- CCBs and CCWs must be located below 16 MB.
- Appendage routines must be below the 16 MB line.
- LIOCS (logical input output control system) uses only format-0 CCWs and data areas are located below the 16 MB line.
- VSE/POWER supports format-0 CCWs only.
- EXCP REAL is supported for format-0 CCWs only.
- A format-1 CCW console I/O is supported but if you use VSE/OCCF console requests they are restricted to format-0 CCWs.

Other Macros

SPLEVEL Macro

The SPLEVEL (set and test macro level) macro sets or tests the global symbol that indicates the level of a macro. The macro is activated at compile time and important for users who want to run their programs on back-level releases.

Specific macros supplied in the macro library are identified as downward incompatible (to VSE/ESA 1.1, 1.2, or 1.3). Unless you take specific action, these macros generate downward incompatible statements.

It is possible to generate downward compatible expansions for some of these macros by using the SPLEVEL macro. Downward incompatible macros interrogate a global symbol (set by SPLEVEL with the SET parameter) during assembly to determine the type of expansion to be generated. For example, the following macros check the setting of the global symbol:

- FCEPGOUT
- PAGEIN

- PFIX
- PFREE
- RELPAG
- WTO
- WTOR

The High Level Assembler reference manuals give information about global set symbols.

The following SET values apply:

- VSE/ESA 1.1 and 1.2 macro expansion if SET=1
- VSE/ESA 1.3 macro expansion if SET=2
- VSE/ESA 1.3 macro expansion if SET=3 (for MVS™ compatibility reasons)
- VSE/ESA 2.1 and later macro expansion if SET=4 (default value)

Refer to “Using the SPLEVEL Macro” on page 13 for an example of how to use the SPLEVEL macro.

STXIT Macro

The STXIT (set exit) macro can enable an exit (AB, IT, OC, PC) and defines a save area where the interrupt information will be stored before exit activation. The STXIT macro supports two kinds of save areas, an old format and an extended format. The old save area is 72 bytes in length. The first 8 bytes contain the interrupt status in the form of a BC mode PSW and the remaining bytes include general registers 0-15.

The BC mode PSW cannot be used in 31-bit addressing mode, because the first byte of the instruction address is required for the instruction length and condition code. Furthermore, user programs can operate in access register mode (AR mode).

The access registers (ARs) may also be of interest for the exit routine. Therefore the exit save area is extended by a field that contains the EC mode PSW and 16 fullwords containing the ARs if the interrupted program operates in AR mode. The extended save area is also required for STXIT OC with MSGDATA=YES and MSGPARM=YES. When using the extended format the instruction address of the BC mode PSW is cleared to zero.

The user-supplied save area specified in the STXIT macro may reside below or above 16 MB. The new AMODE parameter specifies whether the old or extended save area is to be used and where the save area is located.

Note:

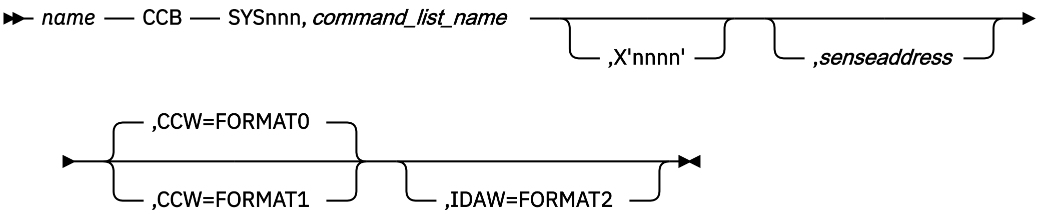
1. The exit routine gets control in the mode specified in the AMODE parameter of the STXIT macro; that is in AMODE 24 for STXIT ...,AMODE=24 and in AMODE 31 for STXIT ...,AMODE=ANY.
2. The old save area layout can be used in 24-bit addressing mode only.

Macro Support for 64-bit Addressing

If a calling program wishes to use 64-bit I/O buffers, it must use a Format-2 IDAW to pass the 64-bit virtual address of the I/O buffer within the Channel Command Word (CCW).

Since the Format-2 IDAW is a 64-bit field that is indicated via the CCB macro, the calling program can also use the Format-2 IDAW to pass a virtual 24-bit or 31-bit address.

The format of the CCB macro is shown in the syntax diagram below.



IDAW=FORMAT2
Indicates that a Format-2 IDAW is to be used.

Note: Format-1 IDAWs are *not* supported by z/VSE.

When Format-2 IDAW is indicated in the corresponding CCB and the IDA (Indirect Data Addressing)-bit in the CCW is set, the CCW data address portion points to a single Format-2 IDAW containing the 64-bit virtual address. This is shown in Figure 35 on page 135.

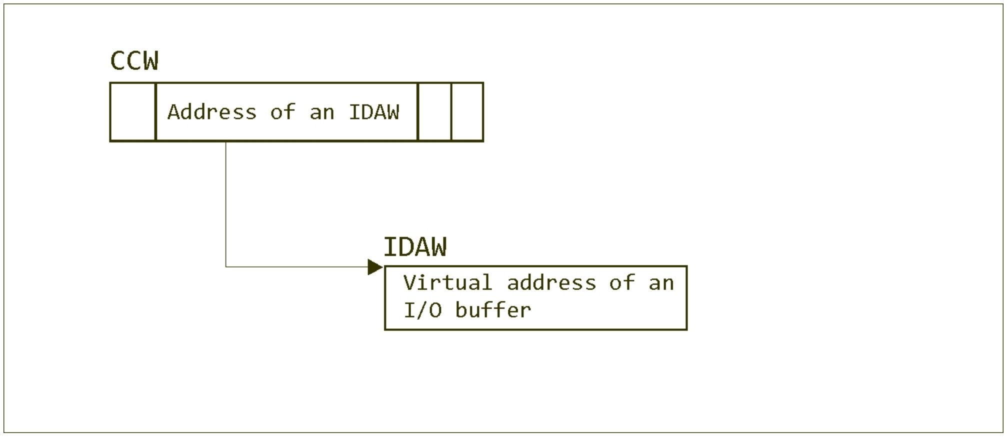


Figure 35. Performing an I/O Request When Using 64-Bit I/O Buffers

For a description of the other CCB macro parameters shown in the above syntax diagram, refer to the topic "CCB (Command Control Block Definition) Macro" in the z/VSE System Macros Reference.

Macro and Command Support for Data Spaces

This section introduces selected macros and commands that are new or that have been enhanced for the support of data spaces. For the syntax and a detailed keyword and parameter description, refer to z/VSE System Macros Reference for the macros and to z/VSE System Control Statements for the commands.

ALESERV Macro

This macro supports a subset of the z/OS macro of the same name and is described in detail in z/VSE System Macros Reference under "ALESERV Macro".

The ALESERV macro controls the entries in the access list. An access list is a table in which each entry identifies an address space or data space to which a program (or programs) has access.

Note: The ALESERV macro cannot be used to control access list entries for address spaces.

Access list entry tokens (ALETs) index the entries in the access list.

On the ALESERV macro, data spaces are identified through STOKENs, an identifier similar to a partition ID. Use the ALESERV macro to:

- Add an entry to the DU-AL or PASN-AL for a SCOPE=SINGLE or a SCOPE=ALL data space (ADD parameter).
- Add an entry to all PASN-ALs for a SCOPE=COMMON data space (ADD parameter).

- Delete an entry from the DU-AL or PASN-AL (DELETE parameter).
- Delete an entry from all PASN-ALs for a SCOPE=COMMON data space (DELETE parameter).
- Obtain a STOKEN for a specified ALET (EXTRACT parameter).
- Locate an ALET for a specified STOKEN (SEARCH parameter).

Restrictions when Using the ALESERV Macro

1. To compile the ALESERV macro, you need the High Level Assembler.
2. VSE/ICCF interactive partitions, system tasks and EXEC `...,REAL` applications are not allowed to call ALESERV services.

An access list entry (ALE) added to the PASN-AL stays in the access list until the partition is deallocated.

Calling Requirements for ALESERV Macro

Authorization:

To request the following ALESERV services, the issuer must have PSW key 0:

- Make ADD and DELETE requests for PASN-AL
- Make ADD and DELETE requests for SCOPE=ALL and SCOPE=COMMON data spaces

All other services can be requested with any PSW key.

ATTACH ALCOPY Macro

The ATTACH ALCOPY macro allows a program to pass a copy of the DU-AL, that belongs to the attaching task, to the subtask to be attached. Refer to “Attaching a Subtask and Sharing Data Spaces with It” on page 96 for an example.

DSPSERV Macro

This macro supports a subset of the z/OS macro of the same name and is described in detail in z/VSE System Macros Reference under "DSPSERV Macro".

The DSPSERV macro creates, deletes and controls data spaces. There are three kinds of data spaces:

- SCOPE=SINGLE
- SCOPE=ALL
- SCOPE=COMMON

A SCOPE=SINGLE data space is used in ways similar to the use of the partition GETVIS area. A SCOPE=ALL or SCOPE=COMMON data space is used in ways similar to the use of the shared virtual area (SVA) of an address space. A program with a non-zero PSW key cannot create or delete a SCOPE=ALL or SCOPE=COMMON data space. However, it can use these spaces, providing a program with PSW key 0 created the data space and established addressability.

Use the DSPSERV macro to:

- Create a data space (CREATE parameter)
- Delete a data space (DELETE parameter)
- Release an area of a data space (RELEASE parameter)
- Increase the current size of a data space (EXTEND parameter)

On the DSPSERV macro, data spaces are identified through STOKENs. A STOKEN is a unique identifier of data spaces.

Restrictions when Using the DSPSERV Macro

1. To compile the DSPSERV macro, you need the High Level Assembler.
2. VSE/ICCF interactive partitions, system tasks, and EXEC ...,REAL applications are not allowed to call DSPSERV services.

The end of task process deletes all data spaces owned by the terminating task.

Calling Requirements for DSPSERV Macro

Authorization:

To request the following DSPSERV services, the issuer must have PSW key 0:

- Create and delete a SCOPE=ALL and SCOPE=COMMON data space
- Extend the current size of a data space it does not own

All other services can be requested with any PSW key.

SDUMPX Macro

The SDUMPX macro dumps address ranges in any data spaces to which addressability via an ALET or via an STOKEN exists. The dump is directed to a dump library or to SYSLST.

SETPFA Macro

The SETPFA macro either sets up or removes linkage to a user-written page-fault appendage routine. With the DSPACE parameter it is possible to specify whether the appendage routine is to process page faults for both address and data spaces or for address spaces only.

SYSSTATE Macro

The SYSSTATE macro is used to set and test a global symbol.

Certain macros that support callers in both access register (AR) and primary address space control (ASC) mode need to know which ASC mode your program is running in.

- The macros that support callers in AR mode generate the code and addresses that are appropriate for callers in AR mode; macros that support callers in primary mode generate the code and addresses that are appropriate for callers in primary mode.
- These macros use the SYSSTATE TEST macro to test a global symbol that is set through the SYSSTATE ASCENV macro.
- The name of the global symbol is &SYSASCE.

It is recommended to issue the SYSSTATE ASCENV=AR macro at the time your program changes ASC mode to AR mode. Then, when your program returns to primary mode, issue SYSSTATE ASCENV=P.

The following is a list of the macros that check the setting of the global symbol:

- ALESERV
- DSPSERV
- SDUMP/SDUMPX

Example for SYSSTATE Macro

To change the ASC mode to AR mode and set the global symbol, issue:

```
SAC      512
SYSSTATE ASCENV=AR
```

SYSDEF Command

The AR/JCL command SYSDEF defines limits and defaults for data spaces such as:

- The total amount of virtual storage that can be allocated to data spaces (the allocated storage is taken from the IPL VSIZE).
- The maximum number of data spaces that can be allocated within the system at one time.
- The maximum number of data spaces that can be allocated per partition at one time.
- The maximum number of data spaces with SCOPE=COMMON that can be allocated at one time.
- The default size of a data space.

QUERY Command

The AR/JCL command QUERY can be used to display data space limits and defaults and further details such as data space names and sizes.

MAP Command

The MAP command can be used to display the amount of virtual storage allocated for data spaces.

Appendix C. Channel Program Support for Virtual Disks

Note: This appendix is intended for such users only who write their own channel programs for virtual disks and must, therefore, know the details provided in the following topics.

In z/VSE, a virtual disk emulates a real FBA disk device. As with a real FBA device, you can use the CCB macro and channel command words (CCWs) to write a channel program for accessing a virtual disk. In general, such a channel program accesses data by using a:

1. **DEFINE EXTENT** command to pass information about the extent of the area (or *space*) on the virtual disk for which subsequent commands are valid.
2. **LOCATE** command to specify a specific address and the amount of data to be transferred.
3. **READ** or **WRITE** command for data transfer.

Both Format-0 and Format-1 CCWs are valid in a channel program. You specify the type used in the CCB macro. Note, however, that:

- The *channel program* must reside below the 16 MB line, whether you use Format-0 or Format-1 CCWs.
- If the *data area* of your channel program is above 16 MB, then you must use Format-1 CCWs.

"I/O Processing Support for 31-Bit Addressing" on page 132 provides details about the two CCW formats.

Channel Commands

Channel commands that can be used with virtual disks are described in the following paragraphs. All other channel commands *are rejected* with Unit Check status and Command Reject indicated in the sense data. In particular, these valid FBA commands are not supported for virtual disks:

- Read Initial Program Load (X'02')
- Read and Reset Buffered Log (X'A4')
- Diagnostic Control (X'F3')
- Diagnostic Sense/Read (X'C4')
- Device Reserve (X'B4')
- Device Release (X'94')
- Unconditional Reserve (X'14')

DEFINE EXTENT (X'63')

As shown in Table 10 on page 140, the Define Extent command operates on 16 bytes of information which define an addressing range on a virtual disk. Subsequent chained commands may operate only within that addressing range. Also included is an operation inhibit mask (byte 0, bits 0-1).

If less than 16 bytes are specified, the command is rejected with Unit Check (Command Reject), Channel End, and Device End. If the CCW count is greater than 16 bytes, only 16 bytes of information are used.

If another Define Extent command was previously issued in the same chain, the command is rejected with Unit Check (Command Reject), Channel End, and Device End.

If parameters of the extent are incorrect, the command is terminated with Unit Check (Command Reject), Channel End and Device End status.

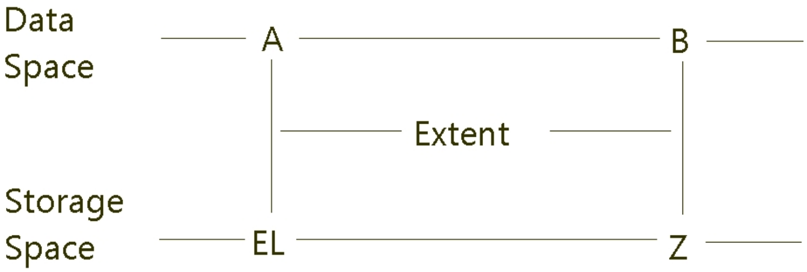
If the addressable block size is incorrect, the command is terminated with Unit Check (Block Size Exception), Channel End and Device End status.

If the parameters are valid and command chaining is not indicated, Channel End and Device End status are presented.

If the parameters are correct and command chaining is indicated, the next CCW is executed. This is normally a Locate command.

Before continuing, two terms must be defined. The first term is **storage space**. Storage space is the usable storage contained in a virtual disk. Storage space is addressed in blocks from 0 to N-1.

The second term is disk **data space**. This is *not* the same as a z/VSE data space, which is used to create a virtual disk. Instead, disk data space refers to space on a virtual disk that contains *addressable blocks of user data*. These blocks are numbered from 0 to M-1. An Extent Locator specified in a Define Extent command references a disk data space to its storage space. An example follows:



In this example, points A (bytes 8-11) and B (bytes 12-15) represent the limits of the extent in the data space. EL is the Extent Locator (bytes 4-7). EL and Z represent the limits of the extent in storage space. All points (A, B, EL, and Z) must be valid block numbers.

For example, EL = 500, A = 150, B = 400.

The location in the storage space of A is 500 and of B it is 750.

The following 16 bytes are the parameters of the Define Extent command.

Table 10. Parameters of the Define Extent command

Bytes	Bits	Mask Byte	Description
0	0-1	00 01 10 11	Non-formatting write permitted All write operations inhibited Reserved - not allowed All write operations permitted
	2-7		Unused - must be zero
1			Reserved - must be zero
2-3			Addressable block size in bytes (512,or 0). 0 defaults to 512.
4-7			Extent Locator. This is a block number which identifies the location on the storage space of the first block of an extent.
8-11			Addressable block number of the first block of this extent in the data space
12-15			Addressable block number of the last block of this extent in the data space

The following describes the bytes used by the Define Extent command.

Byte 0: If any of byte 0 bits 2-7 is not zero or bits 0-1 are B'10', then Unit Check (Command Reject), Channel End, and Device End are set.

Bytes 2-3: The addressable block size specification in bytes 2-3 must be 512 or 0. A block size of 0 is interpreted as a default value of 512.

Bytes 4-7: Bytes 4-7 contain the extent locator. The extent locator is the block number specifying the location on the virtual disk of the first block of an extent.

Bytes 8-11: Bytes 8-11 contain the addressable block number of the first block of data for this extent.

Bytes 12-15: Bytes 12-15 contain the addressable block number of the last block of data for this extent. This value must obey the following rules or the command is rejected:

- A less than or equal to B, and
- D less than or equal to the maximum data block number that is valid for the device.

Where:

A = Value in bytes 8-11 (number of first block of extent)
B = Value in bytes 12-15 (number of last block of extent)
C = Value in bytes 4-7 (extent locator)
D = C+(B-A).

LOCATE (X'43')

The Locate command specifies the addressable block number of the first block of the data space to be processed and the number of sequential blocks to be processed. The specification consists of eight bytes.

If the Locate command is not preceded by a Define Extent command in the same chain, the command is rejected with Unit Check (Command Reject), Channel End, and Device End status.

If less than 8 bytes are specified, the command is rejected with Unit Check (Command Reject), Channel End, and Device End status.

If more than 8 bytes are specified, only 8 bytes are used.

If chaining is not indicated, Channel End and Device End status is presented after a valid 8-byte parameter list is received. The following 8 bytes are the parameters of the Locate command:

Bytes	Bits	Description
0		Operation byte
	0-3	Reserved - must be zero
	4-7	Operation Code: • 0001 Write Data • 0101 Write and Check Data • 0010 Read Replicated Data • 0110 Read Data
1		Replication count
2-3		Block count. This is the number of addressable blocks to be processed.
4-7		Addressable block number of the first block to be processed in the data space

The following describes the 8 bytes used by the Locate command.

Byte 0: Byte 0 predefines the operation to be performed. Bits 4-7 define the specific operation code. Data transfer associated with an operation code does not occur during the execution of the Locate command, but is initiated by the Read or Write CCW that follows the Locate command. Operation codes or modifiers that are not assigned are incorrect and cause the Locate command to be terminated with Unit Check (Command Reject), Channel End, and Device End status.

The following is a description of the operation codes.

- ***Write Data:*** This operation code prepares to write one or more addressable blocks of data. The number of blocks to be written is determined from the block count (bytes 2-3). If the define extent mask inhibits all write operations, the Locate command is rejected with Unit Check (Command Reject), Channel End, and Device End status. A write data operation code establishes write state for the virtual disk.

  If the parameters are valid and command chaining is active, the next CCW is executed.
- ***Write and Check Data:*** Same as ***Write Data***. No checking is done on a virtual disk.
- ***Read Replicate:*** This operation code prepares to read one or more addressable blocks of data from a range of replicated data. The number of blocks to be read is determined from the block count (bytes 2-3). Read Replicated Data establishes read state for the virtual disk.

  If the parameters are valid and command chaining is active, the next CCW is executed.
- ***Read Data:*** This operation code prepares to read one or more addressable blocks of data. The number of blocks to be read is determined from the block count (bytes 2-3). The Read Data operation code establishes read state for the virtual disk.

  If the parameters are valid and command chaining is active, the next CCW is executed.

Byte 1: Byte 1 is the replication count. Byte 1 must be zero if byte 0 specifies Read, Write, or Write and Check Data. When byte 0 specifies Read Replication Data, byte 1 specifies a range of addressable blocks containing replicated data. The first block of the range is specified by the relative block number in bytes 4-7.

The block count (bytes 2-3) specifies the number of addressable blocks in a unit of replicated data. For example, if the block count is two and if this two-block unit is replicated five times, then the replication count is ten.

If the replication count is less than the block count or if the replication count is not an integral multiple of the block count, then the Locate command is terminated with Unit Check (Command Reject), Channel End, and Device End status.

If the replication count is equal to the block count, read replicate is treated as a normal read.

Bytes 2-3: Bytes 2 and 3 contain the block count parameter. This parameter specifies the number of sequential addressable blocks to be processed by the command immediately following the Locate command. A count of zero is rejected with Unit Check (Command Rejected), Channel End, and Device End.

Bytes 4-7: Bytes 4 through 7 specify the data space addressable block of the first block to be processed.

The addressable block numbers of the blocks to be processed are compared against the extent limits established by the preceding Define Extent parameters.

All blocks to be processed must be within the valid extent range. If these conditions are not satisfied, the Locate command is terminated with Unit Check (File Protected), Channel End, and Device End status.

READ (X'42')

The Read command causes the actual data transfer from the virtual disk to main storage. A prior Locate command determines the location from which data is to be transferred.

The Read command must be command-chained from a Locate command. If not, this command is rejected with Unit Check (Command Reject), Channel End, and Device End status.

The Locate command from which the Read command is command-chained must have established read orientation. If not, the Read command is rejected with Unit Check (Command Reject), Channel End, and Device End status.

If command-chaining is not indicated, Channel End and Device End are both presented after the data is successfully read.

For Read commands, a CCW count less than the byte count derived from the block count of the Locate command terminates the data transfer when the CCW count reaches zero. If the CCW count is larger than the block count in the Locate command, the Locate block count terminates the data transfer when it reaches zero.

WRITE (X'41')

The Write command causes data from main storage to be transferred to the virtual disk. A prior Locate command determines the location to which this data is to be written.

The Write command must be command-chained from a prior Locate command. If not, this command is rejected with Unit Check (Command Reject), Channel End, and Device End status.

The Locate command from which the Write command is chained must specify write-orientation. If not, the Write command is rejected with Unit Check (Command Reject), Channel End, and Device End status.

If the CCW count is less than the byte count derived from the block count of the Locate command, data transfer is terminated when the CCW count reaches zero. However, zeros are padded until the Locate block count reaches zero. If the CCW count is greater than the block count, the block count terminates the operation.

If command chaining is not indicated, Channel End and Device End status are both presented after the data is successfully written on the file.

If Write and Check Data was specified in the operation byte of the Locate command, the same as for "Write Data" is performed.

NO-OPERATION (X'03')

The No-Operation command causes no action to be performed.

SENSE (X'04')

The Sense command causes up to 32 bytes of sense data to be transferred to main storage. These bytes identify the specific nature of an error or unusual condition that caused the last Unit Check status to be presented.

If command chaining is not indicated at the completion of the data transfer, Channel End and Device End status are both presented and the sense data is reset to zero.

Note:

1. "Sense Information" on page 145 has details about the information in the sense bytes.
2. When a Unit check occurs, z/VSE normally issues a SENSE CCW automatically to retrieve the sense information. You can specify an address in the CCB macro where z/VSE should pass the sense information to your program. This is recommended, because after a SENSE CCW or after execution of any CCW command, the sense information for that virtual disk is cleared (reset to zero).

TRANSFER IN CHANNEL

CCW Format-0 = **X'X8'**. CCW Format-1 = **X'08'**.

The Transfer In Channel (TIC) command provides the main storage address of the next CCW in the chain. The operation is terminated with Program Check status when:

- The address of the next CCW is not located on a doubleword boundary.
- There are two consecutive TICs in a chain.

SENSE ID (X'E4')

The Sense ID command causes the transfer of a maximum of 7 bytes of I/O identification data to main storage. If the CCW count specifies more than 7 bytes, only 7 bytes are transferred to the system. If the CCW count is less than 7 bytes, only the number of bytes specified is transferred to the system.

The bytes contain the information shown in the following tables.

Bytes	Contents	Description
0	X'FF'	
1-2	X'FBA0'	Control Unit Type
3	X'00'	Control Unit Model
4-5	X'FBA0'	Device Type
6	X'00'	Device emulated

READ DEVICE CHARACTERISTICS (X'64')

The Read Device Characteristics command transfers up to 32 bytes of device specific data to the system. If the CCW count specifies more than 32 bytes, only 32 bytes are transferred. If the CCW count is less than 32 bytes, only the number of bytes specified is transferred.

The following parameters are passed to the requestor.

Byte 0	X'60'	Operation modes
Byte 1	X'28'	Features
Byte 2	X'21'	Device Class
Byte 3	X'00'	Unit type
Bytes 4-5	X'200'	Physical record size in bytes
Bytes 6-9	X'40'	Number of addressable blocks per cyclical group
Bytes 10-13	X'3C0'	Number of addressable blocks per access position
Bytes 14-17	X'nn'	Number of addressable blocks
Bytes 18-23	X'00'	Reserved, set to zero
Bytes 24-25	X'00'	Number of addressable blocks in CE area
Bytes 26-31	X'00'	Reserved, set to zero

If command chaining is not indicated at the completion of the data transfer, Channel End and Device End status are presented.

Note: Instead of writing a channel program to retrieve the device characteristics of a virtual disk, you also can use the **GETVCE** macro, as described in the topic "GETVCE Macro" on page 102.

Flags

Table 11 on page 145 shows which flags are supported, as described in the z/Architecture Principles of Operation manual. In the table, the following meaning applies:

CC =
Chain Command flag

CD =
Chain Data flag

IDA =
Indirect Data flag

SK =
Skip flag

SLI =
Suppress Length Indication flag

PCI =
Program Controlled Interruption flag

S =
Suspend flag

Table 11. Supported CCW Command Flags

CCW Command	CC	CD	IDA	SK	SLI	PCI	S
Define Extent	yes	yes	yes	ignored	yes	ignored	ignored
Locate	yes	yes	yes	ignored	yes	ignored	ignored
Read	yes	yes	yes	yes	yes	ignored	ignored
Write	yes	yes	yes	ignored	yes	ignored	ignored
No-Operation	yes	ignored	ignored	ignored	ignored	ignored	ignored
Sense	yes	yes	yes	yes	yes	ignored	ignored
Transfer in Channel	ignored	ignored	ignored	ignored	ignored	ignored	ignored
Sense ID	yes	yes	yes	yes	yes	ignored	ignored
Read Device Char.	yes	yes	yes	yes	yes	ignored	ignored

Note:

1. In z/VSE, the IDA flag is generally not supported for virtual channel programs. If the data area of a channel program is above 16 MB, then Format-1 CCWs have to be used. You can specify the IDA flag in any CCW *only if* you specify the REAL parameter in your EXCP macro.
2. The PCI and S flags are not supported and thus ignored.

Sense Information

In case of Unit Check, a virtual disk might return sense information containing a fault symptom code. This sense information can be retrieved by a program if sense is specified in the CCB macro. Sense information also is displayed by some messages on the z/VSE system console.

Information Returned to a Sense Command

A Unit Check is raised if the virtual disk detects an unusual situation. The following information is returned to a Sense command:

Bytes	Bit Definition
0	0 = Command Reject (CR)
	1-2 = not set
	3 = Equipment Check
	4-7 = not set

Bytes	**Bit Definition**
1	0 = Permanent Error (PE)
	1 = Block Size Exception (BE)
	2-4 = not set
	5 = File Protected (FP)
	6-7 = not set
2-21	not set
22-23	Fault symptom code (in hexadecimal)
24-31	not set

Note: For any type of I/O error on a page data set where the data space used for a virtual disk resides, "permanent error" and "equipment check" are set in sense bytes 0 and 1.

Fault Symptom Code (Bytes 22-23) of Sense Information

One of the codes below might be passed if the appropriate situation occurs.

Table 12. General Fault Symptom Codes

Sense Bytes 22-23		**Sense Bytes 0-1**	**Explanation**
Decimal **0061**	**Hex.** X'003D'	CR (X'8000')	CCW command code rejected (not supported for a virtual disk)

Table 13. Fault Symptom Codes for DEFINE EXTENT

Sense Bytes 22-23		**Sense Bytes 0-1**	**Explanation**
Decimal **0101**	**Hex.** X'0065'	CR (X'8000')	DEFINE EXTENT was already issued in the CCW chain.
0102	X'0066'	CR	CCW byte count is less than 16.
0103	X'0067'	BE (X'0040')	Block size is neither 0 nor 512.
0104	X'0068'	CR	Parameter byte 0: bits 2, 3, 4, 5, 6, 7 are not all zero.
0105	X'0069'	CR	Parameter byte 0: bits 0-1 are B'10', which is not allowed.
0106	X'006A'	CR	Parameter byte 1: not zero.
0107	X'006B'	CR	Parameter bytes 4-7: first block of storage space is larger than last block of storage space.
0108	X'006C'	CR	First block of data space (parameter bytes 8-11) is larger than last block of data space (parameter bytes 12-15).
0109	X'006D'	CR	First block of storage space (parameter bytes 4-7) plus size of data space (parameter bytes 12-15 minus bytes 8-11) is larger than last block of storage space.

Table 14. Fault Symptom Codes for LOCATE

Sense Bytes 22-23		Sense Bytes 0-1	Explanation
Decimal **0151**	**Hex.** X'0097'	CR (X'8000')	No DEFINE EXTENT was previously issued.
0152	X'0098'	CR	CCW byte count is less than 8.
0153	X'0099'	CR	Parameter byte 0: bits 0, 1, 2, 3 not all zero
0154	X'009A'	CR	Parameter byte 0: bits 4-7 are not B'0001', B'0010', B'0101', or B'0110'.
0155	X'009B'	CR	Parameter byte 0: bits 4-7 do not specify either B'0010' (read replicate data) or B'0110' (read). This causes LOCATE to be write-oriented. However, parameter byte 0 (bits 0-1 of the previous DEFINE EXTENT) are set to B'01', which means "inhibit all write operations".
0156	X'009C'	CR	Parameter bytes 2-3: (block count) are zero.
0157	X'009D'	CR	The read replicate function was not specified (parameter byte 0, bits 4-7), but the replication count (parameter byte 1) is not zero.
0158	X'009E'	CR	The read replicate function was specified (parameter byte 0, bits 4-7), and the replication count (parameter byte 1) is less than the block count (parameter bytes 2-3)
0159	X'009F'	CR	The replication count is not an integral multiple of the block count.
0160	X'00A0'	FP (X'0004')	The first block of the data space to be processed (parameter bytes 4-7) is less than the first block of the data space specified in the DEFINE EXTENT parameter (bytes 8-11).
0161	X'00A1'	FP	The first block of the data space to be processed (parameter bytes 4-7) is larger than the last block of the data space specified in the DEFINE EXTENT parameter (bytes 12-15).
0162	X'00A2'	FP	The last block of the data space to be processed (first block to be processed plus block count) is larger than the last block of the data space specified in the DEFINE EXTENT parameter (bytes 12-15).

Table 15. Fault Symptom Codes for READ

Sense Bytes 22-23		Sense Bytes 0-1	Explanation
Decimal **0201**	**Hex.** X'00C9'	CR (X'8000')	Not chained from LOCATE.
0202	X'00CA'	CR	Previous LOCATE command was not read-oriented.
0685	X'02AD'	PE (X'1080')	Unrecoverable I/O error on page data set in data space.
0686	X'02AE'	PE	Unrecoverable I/O error on page data set in data space.
0687	X'02AF'	PE	Unrecoverable I/O error on page data set in data space.

Table 16. Fault Symptom Codes for WRITE

Sense Bytes 22-23		**Sense Bytes 0-1**	**Explanation**
Decimal **0221**	**Hex.** X'00DD'	CR (X'8000')	Not chained from LOCATE.
0222	X'00DE'	CR	Previous LOCATE command was not write-oriented.
0665	X'0299'	PE (X'1080')	Unrecoverable I/O error on page data set in data space.
0666	X'029A'	PE	Unrecoverable I/O error on page data set in data space.
0667	X'029B'	PE	Unrecoverable I/O error on page data set in data space.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information 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 actual people or business enterprises 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.

Programming Interface Information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/VSE.

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" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IPv6/VSE is a registered trademark of Barnard Software, Inc.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein. IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed. You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/VSE enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using Assistive Technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/VSE. Consult the assistive technology documentation for specific information when using such products to access z/VSE interfaces.

Documentation Format

The publications for this product are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF files and want to request a web-based format for a publication, you can either write an email to s390id@de.ibm.com, or use the Reader Comment Form in the back of this publication or direct your mail to the following address:

```
IBM Deutschland Research & Development GmbH
Department 3282
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany
```

In the request, be sure to include the publication number and title.

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

Glossary

If you do not find the term you are looking for, refer to the index of this book or to the *IBM Dictionary of Computing* New York: McGraw Hill, 1994.

The glossary includes definitions with:

- Symbol * where there is a one-to-one copy from the IBM Dictionary of Computing.
- Symbol (A) from the *American National Dictionary for Information Processing Systems* , copyright 1982 by the Computer and Business Equipment Manufacturers Association (CBEMA). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- Symbols (I) or (T) from the *ISO Vocabulary - Information Processing* and the *ISO Vocabulary - Office Machines,* developed by the International Organization for Standardization, Technical Committee 97, Subcommittee 1. Definitions of published segments of the vocabularies are identified by the symbol (I) after the definition; definitions from draft international standards, draft proposals, and working papers in development by the ISO/TC97/SC1 vocabulary subcommittee are identified by the symbol (T) after the definition, indicating final agreement has not yet been reached among participating members.

The following cross-references are used:

- Contrast with. This refers to a term that has an opposed or substantively different meaning.
- Synonym for. This indicates that the term has the same meaning as a preferred term, which is defined in its proper place in the dictionary.
- Synonymous with. This is a backward reference from a defined term to all other terms that have the same meaning.
- See. This refers the reader to multiple-word terms that have the same last word.
- See also. This refers the reader to related terms that have a related, but not synonymous, meaning.

When an entry is an abbreviation, the explanation consists of the spelled-out meaning of the abbreviation, for example:

```
AFP. Advanced Function Printing.
```

The spelled-out form is provided as a separate entry in the glossary. In that entry, the abbreviation is shown in parentheses after the spelled-out form. The definition that appears with the spelled-out entry provides the full meaning of both the abbreviation and the spelled-out form:

```
Advanced Function Printing (AFP). A group of...
```

access list

A table in which each entry specifies an address space or data space that a program can reference.

access method

A program, that is, a set of commands (macros), to define files or addresses and to move data to and from them; for example VSE/VSAM or VTAM®.

access register (AR)

A hardware register that a program can use to identify an address space or a data space. Each processor has 16 ARs, numbered 0 through 15, which are paired one-to-one with the 16 general-purpose registers (GPRs).

addressing mode (AMODE)

A program attribute that refers to the address length that a program is prepared to handle on entry. Addresses may be either 24 bits or 31 bits in length. In 24-bit addressing mode, the processor treats all virtual addresses as 24-bit values; in 31-bit addressing mode, the processor treats all virtual addresses as 31-bit values. Programs with an addressing mode of ANY can receive control in either 24-bit or 31-bit addressing mode.

address space
A range of up to two gigabytes of contiguous virtual storage addresses that the system creates for a user. Unlike a data space, an address space 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 (not a data space). Contrast with *data space*.

address space control (ASC) mode
The mode (determined by the PSW) that tells the system where to find referenced data. It determines how the processor resolves address references for the executing programs. z/VSE supports two types of ASC modes:

1. In **primary** ASC mode, the data that a program can access resides in the program's own (primary) address space. In this mode, the system uses the contents of general-purpose registers to resolve an address in the address space; it does not use the contents of the access registers (ARs).
2. In **access register (AR)** ASC mode, the data that a program can access may reside in an address space other than the primary or in a data space. In this mode, the system uses both a general-purpose register (GPR) and the corresponding access register together to resolve an address in another address space or in a 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 space or data space.

ALET (access list entry token)
A token that points to an entry in an access list. When a program is in AR mode and the ALET is in an access register (with the corresponding general-purpose register being used as base register), the ALET identifies the address space or data space that the system is to reference (while the GPR indicates the offset within the space).

AR/GPR
Access register and general-purpose register pair.

AR (access register) mode
If a program runs in AR mode, the system uses the access register/general-purpose register pair to resolve an address in an address space or data space. Contrast with *primary mode*. See also *address space control (ASC) mode*.

ASC mode
Address space control mode.

*** assembler**
A computer program that converts assembly language instructions into object code.

assembler language
A programming language whose instructions are usually in one-to-one correspondence with machine instructions and allows to write macros.

attention routine
A routine of the system that receives control when the operator presses the Attention key. The routine sets up the console for the input of a command, reads the command, and initiates the system service requested by the command.

*** auxiliary storage**
All addressable storage, other than main storage, that can be accessed by means of an input/ouput channel; for example storage on magnetic tape or direct access devices. Synonymous with *external storage*.

block
Usually, a block consists of several records of a file that are transmitted as a unit. But if records are very large, a block can also be part of a record only. With FBA disk devices, a block is a string of 512 bytes of data.

*** catalog**
1. A directory of files and libraries, with reference to their locations. A catalog may contain other information such as the types of devices in which the files are stored, passwords, blocking factors. (I) (A) 2. To store a library member such as a phase, module, or book in a sublibrary.

*** cataloged procedure**
A set of control statements placed in a library and retrievable by name.

cell pool
An area of virtual storage obtained by an application program and managed by the callable cell pool services. A cell pool is located in an address space or a data space and contains an anchor, at least one extent, and any number of cells of the same size.

*** chaining**
A logical connection of sublibraries to be searched by the system for members of the same type; for example, phase or object modules.

*** channel command word (CCW)**
A doubleword at the location in main storage specified by the channel address word. One or more CCWs make up the channel program that directs data channel operations.

*** channel program**
One or more channel command words that control a sequence of data channel operations. Execution of this sequence is initiated by a single start I/O (SIO) instruction.

*** compile**
To translate a source program into an executable program (an object program). See also *assembler*.

compiler
A program used to compile.

component
1. Hardware or software that is part of a computer system. 2. A functional part of a product, identified by a component identifier. 3. In VSE/VSAM, a named, cataloged group of stored records, such as the data component or index component of a key-sequenced file or alternate index.

*** configuration**
The devices and programs that make up a system, subsystem, or network.

control block
An area within a program or a routine defined for the purpose of storing and maintaining control information.

*** data management**
A major function of the operating system. It involves organizing, storing, locating, and retrieving data.

data set
See *file*.

data space
A range of up to two gigabytes of contiguous virtual storage addresses that a program can directly manipulate through z/Architecture instructions. Unlike an address space, a data space can hold only user data; it does not contain shared areas, system data or programs. Instructions do not execute in a data space, although a program can reside in a data space as non-executable code. Contrast with *address space*.

default value
A value assumed by the program when no value has been specified by the user.

*** device address**
1. The identification of an input/output device by its channel and unit number. 2. In data communication, the identification of any device to which data can be sent or from which data can be received.

*** device class**
The generic name for a group of device types; for example, all display stations belong to the same device class.

*** device type code**
The four- or five-digit code to be used for defining an I/O device to a computer system.

*** dialog**

1. In an interactive system, a series of related inquiries and responses similar to a conversation between two people. 2. For z/VSE, a set of panels that can be used to complete a specific task; for example, defining a file.

directory

1. A table of identifiers and references to the corresponding items of data. (I) (A) 2. In VSE, specifically, the index for the program libraries. See also *library directory* and *sublibrary directory*.

DU-AL (dispatchable unit - access list)

The access list that is associated with a z/VSE main task or subtask. A program uses the DU-AL associated with its task and the PASN-AL associated with its partition. See also *PASN-AL*.

dynamic partition

A partition created and activated on an 'as needed' basis that does not use fixed static allocations. After processing, the occupied space is released. Contrast with *static partition*.

emulation

The use of programming techniques and special machine features that permit a computer system to execute programs written for another system or for the use of I/O devices different from those that are available.

extended addressability

1. See *31-bit addressing*. 2. The ability of a program to use virtual storage that is outside the address space in which the program is running. Generally, instructions and data reside in a single address space - the primary address space. However, a program can have data in address spaces other than the primary or in data spaces. (The instructions remain in the primary address space, whilst the data can reside in another address space or in a data space.) To access data in other address spaces, a program must use access registers (ARs) and execute in access register mode (AR mode).

extent

Continuous space on a disk or diskette occupied by or reserved for a particular file or VSAM data space.

external storage

Storage that is not part of the processor.

FBA disk device

Fixed-block architecture disk device. A block contains 512 bytes of data.

*** file**

A named set of records stored or processed as a unit. (T) Synonymous with *data set*.

*** generate**

To produce a computer program by selecting subsets of skeletal code under the control of parameters. (A)

High Level Assembler for VSE

A programming language providing enhanced assembler programming support. It is a base program of z/VSE.

*** initial program load (IPL)**

The process of loading system programs and preparing the system to run jobs.

*** input/output control system (IOCS)**

A group of routines provided by IBM for handling transfer of data between main storage and auxiliary storage devices.

interactive

A characteristic of a program or system that alternately accepts input and then responds. An interactive system is conversational, that is, a continuous dialog exists between user and system. Contrast with *batch*.

interface

A shared boundary between two hardware or software units, defined by common functional or physical characteristics. It might be a hardware component or a portion of storage or registers accessed by several computer programs.

job step
One of a group of related programs complete with the JCL statements necessary for a particular run. Every job step is identified in the job stream by an EXEC statement under one JOB statement for the whole job.

*** librarian**
The set of programs that maintains, services, and organizes the system and private libraries.

library
See *VSE library* and *VSE/ICCF library*.

*** library directory**
The index that enables the system to locate a certain sublibrary of the accessed library.

*** library member**
The smallest unit of data that can be stored in and retrieved from a sublibrary.

*** linkage editor**
A program used to create a phase (executable code) from one or more independently translated object modules, from one or more existing phases, or from both. In creating the phase, the linkage editor resolves cross references among the modules and phases available as input. The program can catalog the newly built phases.

linkage stack
An area of protected storage that the system gives to a program to save status information in case of a branch or a program call.

link-edit
To create a loadable computer program by having the linkage editor process compiled (assembled) source programs.

*** lock file**
In a shared disk environment under VSE, a system file on disk used by the sharing systems to control their access to shared data.

logical record
A user record, normally pertaining to a single subject and processed by data management as a unit. Contrast with *physical record* which may be larger or smaller.

*** main task**
The main program within a partition in a multiprogramming environment.

*** megabyte (MB)**
1 024 KB or 1 048 576 bytes.

*** member**
The smallest unit of data that can be stored in and retrieved from a sublibrary. See also *library member*.

message
1. In VSE, a communication sent from a program to the operator or user. It can appear on a console, a display terminal or on a printout. 2. In telecommunication, a logical set of data being transmitted from one node to another.

*** module**
A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine. (A)

MVS/ESA (Multiple Virtual Storage/Enterprise Systems Architecture)
An IBM program providing operating system support.

object module (program)
A program unit that is the output of an assembler or compiler and is input to a linkage editor.

page data set (PDS)
One or more extents of disk storage in which pages are stored when they are not needed in processor storage.

page frame
An area of processor storage that can contain a page.

partition
A division of the virtual address area available for running programs. See also *dynamic partition, static partition*.

PASN-AL (primary address space number - access list)
The access list that is associated with a partition. A program uses the PASN-AL associated with its partition and the DU-AL associated with its task (work unit). See also *DU-AL*.

Each partition has its own unique PASN-AL. All programs running in this partition can access data spaces through the PASN-AL. Thus a program can create a data space, add an entry for it in the PASN-AL, and obtain the ALET that indexes the entry. By passing the ALET to other programs in the partition, the program can share the data space with other programs running in the same partition.

*** physical record**
The amount of data transferred to or from auxiliary storage. Synonymous with *block*.

primary address space
In z/VSE, the address space where a partition is currently executed. A program in primary mode fetches data from the primary address space.

primary mode
If a program runs in primary mode, the system resolves all addresses within the current (primary) address space. Contrast with *AR (access register) mode*. See also *address space control (ASC) mode*.

priority
A rank assigned to a partition or a task that determines its precedence in receiving system resources.

private area
The part of an address space that is available for the allocation of private partitions. Its maximum size can be defined during IPL. Contrast with *shared area*.

procedure
See *cataloged procedure*.

*** processing**
The performance of logical operations and calculations on data, including the temporary retention of data in processor storage while this data is being operated upon.

*** processor**
In a computer, a functional unit that interprets and executes instructions. A processor consists of at least an instruction control unit and an arithmetic and logic unit. (T)

processor storage
The storage contained in one or more processors and available for running machine instructions. Synonymous with *real storage*.

real address
The address of a location in processor storage.

real storage
See *processor storage*.

record
A set of related data or words, treated as a unit. See *logical record*, *physical record*.

residency mode (RMODE)
A program attribute that refers to the location where a program is expected to reside in virtual storage. RMODE 24 indicates that the program must reside in the 24-bit addressable area (below 16 megabytes), RMODE ANY indicates that the program can reside anywhere in 31-bit addressable storage (above or below 16 megabytes).

*** restore**
To write back onto disk data that was previously written from disk onto an intermediate storage medium such as tape.

*** routine**
A program, or part of a program, that may have some general or frequent use. (T)

shared area
An area of storage that is common to all address spaces in the system. z/VSE has two shared areas:

1. The shared area (24 bit) is allocated at the start of the address space and contains the supervisor, the SVA (for system programs and the system GETVIS area), and the shared partitions.
2. The shared area (31 bit) is allocated at the end of the address space and contains the SVA (31 bit) for system programs and the system GETVIS area.

*** shared virtual area (SVA)**
A high address area that contains a system directory list (SDL) of frequently used phases, resident programs that can be shared between partitions, and an area for system support.

static partition
A partition, defined at IPL time and occupying a defined amount of virtual storage that remains constant. Contrast with *dynamic partition*.

STOKEN (space token)
An eight-byte identifier of a data space. It is generated by the system when you create a data space.

sublibrary
A subdivision of a library. Members can only be accessed in a sublibrary.

sublibrary directory
An index for the system to locate a member in the accessed sublibrary.

*** subsystem**
A secondary or subordinate system, usually capable of operating independently of, or asynchronously with, a controlling system. (T)

subtask
A task that is initiated by the main task or by another subtask.

*** supervisor**
The part of a control program that coordinates the use of resources and maintains the flow of processor operations.

supervisor mode
See *ESA mode*.

*** system console**
A console, usually equipped with a keyboard and display screen for control and communication with the system.

system directory list (SDL)
A list containing directory entries of frequently-used phases and of all phases resident in the SVA. The list resides in the SVA.

*** throughput**
1. A measure of the amount of work performed by a computer system over a given period of time, for example, number of jobs per day. (I) (A) 2. In data communication, the total traffic between stations per unit of time.

*** user exit**
A programming service provided by an IBM software product that may be requested during the execution of an application program for the service of transferring control back to the application program upon the later occurrence of a user-specified event.

virtual address
An address that refers to a location in virtual storage. It is translated by the system to a processor storage address when the information stored at the virtual address is to be used.

virtual disk
A range of up to two gigabytes of contiguous virtual storage addresses that a program can use as workspace. Although the virtual disk exists in storage, it appears as a real FBA disk device to the user

program. All I/O operations directed to a virtual disk are intercepted and the data to be written to, or read from, the disk is moved to or from a data space.

Like a data space, a virtual disk can hold only user data; it does not contain shared areas, system data or programs. Unlike an address space or a data space, data is not directly addressable on a virtual disk. To manipulate data on a virtual disk, the program has to perform I/O operations.

virtual storage
Addressable space image for the user from which instructions and data are mapped into processor (real) storage locations.

volume
A data carrier that is mounted and demounted as a unit, for example, a reel of tape or a disk pack. (I) Some disk units have no demountable packs. In that case, a volume is the portion available to one read/write mechanism.

volume ID
The volume serial number, which is a number in a volume label assigned when a volume is prepared for use by the system.

VSE (Virtual Storage Extended)
A system that consists of a basic operating system and any IBM supplied and user-written programs required to meet the data processing needs of a user. Its current version is called z/VSE.

VSE/ESA (VSE/Enterprise Systems Architecture)
The most advanced VSE system currently available.

VSE/ICCF library
A file composed of smaller files (libraries) including system and user data which can be accessed under the control of VSE/ICCF.

VSE library
A collection of programs in various forms and storage dumps stored on disk. The form of a program is indicated by its member type such as source code, object module, phase, or procedure. A VSE library consists of at least one sublibrary which can contain any type of member.

VSE/VSAM (VSE/Virtual Storage Access Method)
An IBM access method for direct or sequential processing of fixed and variable length records on disk devices.

z/Architecture
An IBM architecture for mainframe computers and peripherals. The zSeries family of servers uses the z/Architecture. It is the successor to the S/390® and 9672 family of servers.

z/OS®
An IBM mainframe operating system that uses 64-bit real storage.

31-bit addressing
Provides addressability for address spaces of up to 2 gigabytes (GB). The maximum amount of addressable storage in previous systems was 16 megabytes (MB).

Index

Product Number: 5686-VS6

SC34-2630-02