z/OS

MVS Programming: Authorized Assembler Services Guide

Version 2 Release 1

SA23-1371-02

Note

Before using this information and the product it supports, read the information in "Notices" on page 853.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 10. Virtual storage management. 213

Chapter 11. Accessing the scheduler work area 245

Chapter 12. The virtual lookaside facility (VLF) 251

Chapter 13. Data-in-virtual 267

Chapter 14. Sharing application data (name/token callable services) 293

Chapter 15. Processor storage management. 301

Chapter 16. Sharing data in virtual storage (IARVSERV macro) 309

Chapter 17. The nucleus. 319

Chapter 18. Providing recovery. . . . 327

Chapter 19. Processing program interruptions (SPIE, ESPIE) 413

Chapter 20. Dumping virtual storage (SDUMPX, SDUMP, and IEATDUMP macros). 417

Chapter 21. Protecting the system 441

Chapter 22. Exit routines 461

Chapter 23. User-written SVC routines 483

Chapter 24. Accessing unit control blocks (UCBs) 495

Chapter 25. Dynamic allocation. . . . 521

Chapter 26. Requesting dynamic allocation functions 543

Chapter 27. Dynamic output 719

Chapter 28. Scheduler JCL facility (SJF) 739

Chapter 29. Processing user trace entries in the system trace table . . . 765

Chapter 30. Using system logger services. 771

Figures

Tables

About this information

This information describes the authorized services that the MVS™ operating system provides; that is, services available only to authorized programs. An authorized program must meet one or more of the following requirements:

- Running in supervisor state
- Running under PSW key 0-7
- Running APF authorized.

Some of the services included in this information are not authorized, but are included because they are of greater interest to the system programmer than to the general applications programmer. The functions of these services are of such a nature that their use should be limited to programmers who write authorized programs. Services are also included if they have one or more authorized parameters — parameters available only to authorized programs.

To use an MVS service, the program issues a macro. The following set of companion documents provides the detailed information necessary to code the macros.

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

Some of the topics described in this document are also described in *z/OS MVS Programming: Assembler Services Guide*, *z/OS MVS Programming: Assembler Services Reference ABE-HSP* and *z/OS MVS Programming: Assembler Services Reference IAR-XCT*. However, those documents do not include the authorized services and macros.

Who should use this information

This information is for the programmer who is using assembler language to code a systems program, and who needs to become familiar with the MVS operating system and the services that programs running under it can invoke.

Programmers using this information should have a knowledge of the computer, as described in *z/Architecture: Principles of Operation*, as well as a knowledge of assembler language programming.

System macros require High Level Assembler. Assembler language programming is described in the following information:

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

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

How to use this information

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

z/OS information

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

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

To find the complete z/OS® library, go to IBM Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html).
3. Mail the comments to the following address:

   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   US
4. Fax the comments to us, as follows:

   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:

  z/OS V2R1.0 MVS Authorized Assembler Services Guide
  SA23-1371-02
- The topic and page number that is related to your comment.
- The text of your comment.

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

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

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at z/OS support page (http://www.ibm.com/systems/z/support/).

Summary of changes

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

Summary of changes for z/OS Version 2 Release 1 (V2R1) as updated February 2015

The following changes are made for z/OS Version 2 Release 1 (V2R1), as updated February 2015. In this revision, all technical changes for z/OS V2R1 are indicated by a vertical line to the left of the change.

New

- A new section has been added, "Vector Register protocol for user exits" on page 474.

Changed

- Information about the disabled interrupt exit (DIE) has been updated in "DIE characteristics" on page 468 and "Exit from the DIE routine" on page 468.
- Information has been updated in "SYMLIST on a DD statement - Key = '802B'" on page 663.

Summary of changes in z/OS Version 2 Release 1 (V2R1), as updated December 2013

The following changes are made for z/OS Version 2 Release 1 (V2R1), as updated December 2013. In this revision, all technical changes for z/OS V2R1 are indicated by a vertical line to the left of the change.

New

- New DALDSNV and DALMAXG dsname allocation text units. For details, see "Dsname allocation text units" on page 631.

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Chapter 1. Introduction

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

Most services described in this information can be used by authorized programs only. An authorized program is a program that runs with APF authorization, or runs in supervisor state, or runs with a PSW key of 0-7.

Some of the services can be used by any program, including those that run without APF authorization, or in problem program state, or with a PSW key of 8-F. Unless your program is authorized, you should use these services only if you have the approval of your installation‘s management.

The services, which you invoke by issuing macros or callable services, are described in the following information:

Subtask Creation and Control:
Occasionally, you can have your program executed faster and more efficiently by dividing parts of it into subtasks that compete with each other and with other tasks for execution time. This topic includes information about task creation, using an internal START, and communication with a problem program.

Program Management:
You can use the system to aid communication between segments of a program. This topic includes information about the residency and addressing mode of a module, loading a module, synchronous exits, checkpoint/restart, and re-entrant modules.

Serialization:
Portions of some tasks depend on the completion of events in other tasks, which requires planned task synchronization. Planning is also required when more than one program uses a serially reusable resource. Locking, the must-complete function, shared direct access storage devices, waiting for an event to complete, and indicating event completion are discussed in this topic. This topic also includes information about the global resource serialization latch manager, which authorized programs can use to serialize resources within an address space or within a single MVS system.

Reporting System Characteristics:
This topic describes collecting information about resources and their requestors, using the SRM reporting interface, and obtaining dispatchability data about address spaces.

Tracing Applications Using Component Trace:
Component trace provides common services for defining and collecting diagnostic information about components, subsystems, or applications.

Communication:
This information discusses several different forms of communication:

- Interprocessor communication
- Writing operator messages

- Issuing operator commands
- Routing operator commands in a sysplex environment
- Extended MCS consoles.

Listening for System Events:

The event notification facility (ENF) allows an authorized program to listen for the occurrence of a specific system event. For example, a program could listen for a software record being written to the logrec data set. When the system writes the record to the logrec data set, control passes to a specified listener user exit routine. For a list of the system events that an authorized program can listen for, see "ENF event codes and meanings" on page 168.

Using a Service Request Block (SRB):

Asynchronous inter-address space communication is available through the use of an SRB and its associated SRB routine. This topic describes how you use the macros that initiate the running of an SRB routine and provide its recovery. It also describes how you can serialize the processing of an SRB routine and how to transfer control from an SRB to a TCB.

Virtual Storage Management:

Virtual storage allows you to write large programs without the need for complex overlay structures. This topic describes how to use the GETMAIN, FREEMAIN, and STORAGE macros to allocate and free virtual storage. It also includes descriptions of the VSM functions, available through the use of the VSMLIST, VSMLOC, and VSMREGN macros.

Accessing the Scheduler Work Area (SWA):

The SWA contains information about jobs that are currently in the system. This topic explains how to write a program that can access the information that the system stores in the SWA.

Virtual Lookaside Facility (VLF):

VLF is intended for use by authorized subsystems or major applications that perform repeated DASD operations on behalf of end users. VLF is a set of services that provide an alternate fast-path method of making frequently used, named data objects available to many users. This topic will help you decide whether to use VLF and will help you plan your use of the VLF macros.

Using the Cross-System Coupling Facility (XCF):

This topic, which contains information about the cross-system coupling facility, now appears in *z/OS MVS Programming: Sysplex Services Guide*.

Data-In-Virtual:

Data-in-virtual enables you to create, read, or update external storage data without the traditional GET and PUT macros. The data, which is not broken up into individual records, appears in your virtual storage as a string of contiguous bytes. You can access the data by using any instruction that you normally use to access data in your virtual storage. This technique also provides better performance than the traditional access methods for many applications.

Sharing Application Data (Name/Token Callable Services):

Name/token callable services allow a user to share data between two programs running under the same task, or between two or more tasks or address spaces. This topic includes descriptions of the levels of name/token pairs, ownership and deletion of the pairs, description of the levels for the cross memory uses of the name/token services, and an example of using the services.

Processor Storage Management:
The system administers the use of processor storage (that is, central and expanded storage) and directs the movement of virtual pages among auxiliary, expanded, and central storage in page size blocks. The services provided allow you to release virtual storage contents, load virtual storage areas into central storage, make virtual storage pages read-only or modifiable, and page out virtual storage areas from central storage.

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

The Nucleus:
This topic includes descriptions of the functions available through the use of the DATOFF and NUCLKUP macros.

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

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

Dumping Virtual Storage (SDUMPX and SDUMP Macros):
This topic describes how to obtain SVC dumps using the SDUMPX and SDUMP macros. It includes deciding when to request a dump, using the SYS1.DUMPnn data set, defining the contents of the dump, and suppressing duplicate dumps.

Protecting the System:
This topic includes the maintenance of system integrity, the use of the authorized programming facility, the use of the resource access control facility, changing system status, and protecting low storage.

Exit Routines:
This topic contains information about asynchronous exit routines, timer disabled interrupt exit routines, and dynamic exits services. Dynamic exits services allow you to define exits and control their use within a program using the CSVDYNEX macro.

User-Written SVC Routines:

This topic contains information needed to write SVC routines. It includes the characteristics of the SVC routines, program conventions for SVC routines, and ways to insert SVC routines into the control program.

Accessing Unit Control Blocks (UCBs):
This topic describes how to obtain and use information about unit control blocks (UCBs) and about configuration changes in the system.

Dynamic Allocation:
This topic describes the functions provided by dynamic allocation. It describes installation options you can use to control the processing of dynamic allocation.

Requesting Dynamic Allocation Functions:
The topic includes a description of the parameter list used to request dynamic allocation functions, the return codes, error codes, and information codes.

Dynamic Output:
This topic explains how to use the JES2 and JES3 4.2.1 or later dynamic output facility. This facility enables a program to dynamically describe a system output data set's processing options and to dynamically allocate that data set.

Scheduler JCL Facility (SJF):
This topic describes how to use the Scheduler JCL Facility to manipulate text unit data that represents processing options for system output (sysout) data sets.

Processing User Trace Entries in the System Trace Table:
This topic describes formatting a USRn trace table entry, interpreting return codes from the formatter, and handling errors.

Using the Unit Verification Service:
This topic describes using the unit verification service to obtain information from the eligible device table.

Note: IMPORTANT -------- READ THIS

Keep in mind how the information uses the following terms:

- The term **registers** means general purpose registers. If general purpose registers might be confused with other kinds of registers (such as access registers), the information uses the longer term **general purpose registers**.
- Unless otherwise specified, the **address space control (ASC) mode** of a program is primary mode.
- Many MVS macro services support callers in both primary ASC and access register (AR) ASC mode. Some of these services provide two macros that perform the same function; one macro can generate code and addresses appropriate for callers in primary mode, and the other macro can generate code and addresses appropriate for callers in AR mode.

  The names of the two macros are the same, except the macro that supports the AR mode caller ends with an “X”. These pairs of macros are called **X-macros**. SNAPX is an example of an X-macro. SNAP is for primary mode callers and SNAPX is for primary or AR mode callers. As you use the X-macros, keep in mind the following information:

  – A parameter or function might apply only to the X-macro. (For example, the DSPSTOR parameter is valid only on SNAPX, not SNAP.)

  – Most of the parameters are available on both the X-macro and the non-X-macro.

Macro descriptions in the *z/OS MVS Programming: Assembler Services Reference ABE-HSP* and the *z/OS MVS Programming: Assembler Services Reference IAR-XCT* help you use the X-macros correctly.

Chapter 2. Subtask creation and control

The control program creates a task when it initiates execution of the job step; this task is the job step task. You can create additional tasks in your program. If you do not, however, the job step task is the only task in a job being executed. The benefits of a multiprogramming environment are still available even with only one task in the job step; work is still being performed for other jobs when your task is waiting for an event, such as an input operation, to occur.

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

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

You can also use tasks in enclaves. An enclave is a transaction that can span multiple tasks and/or SRBs in one or more address spaces and is managed and reported as a unit. A task can join or leave an enclave. While a task is joined to an enclave, any subtasks created also belong to the enclave. For more information about using tasks in enclaves, see *z/OS MVS Programming: Workload Management Services*.

Most of the information concerning subtask creation and control appears in *z/OS MVS Programming: Assembler Services Guide*, *z/OS MVS Programming: Assembler Services Reference ABE-HSP* and *z/OS MVS Programming: Assembler Services Reference IAR-XCT*. This topic provides additional information about:

- Creating a task (ATTACH or ATTACHX macro)
- Ensuring that a process completes (STATUS macro)
- Communicating with a problem program (EXTRACT and QEDIT macros)

Creating a new task (ATTACH or ATTACHX macro)

To create a new task, use the ATTACH or ATTACHX macro. Programs running in primary ASC mode or programs running in access register ASC mode can use either ATTACH or ATTACHX. IBM® recommends, however, that programs running in access register ASC mode use ATTACHX. The ATTACH and ATTACHX services for programs that run in problem program state are described in the *z/OS MVS Programming: Assembler Services Guide* and the *z/OS MVS Programming: Assembler Services Reference ABE-HSP*.

When you attach a subtask, you have the option of setting its dispatchability. The DISP parameter of the ATTACH and ATTACHX macros enables you to set the attached subtask to be dispatchable (DISP=YES) or non-dispatchable (DISP=NO). You can reset tasks that you attached with the DISP=NO parameter to dispatchable by issuing ATTACH or ATTACHX again with the DISP=RESET parameter.

Ensuring that a process completes (STATUS macro)

Complex programs, such as database managers, often include processes that use critical resources and that must complete quickly. While it is possible to prevent delays by running the program disabled, disabled programs have certain restrictions, such as not being able to perform I/O and not being able to reference pageable storage. If your program cannot run with the restrictions of a disabled program, and it cannot tolerate a process delay, issue the STATUS macro with the SET,MC,PROCESS parameter, taking into consideration the restrictions discussed later in this topic. Issuing STATUS SET,MC,PROCESS puts the program in process-must-complete mode.

Issuing STATUS SET,MC,PROCESS at the start of a process postpones delays from certain events. The macro allows a process using critical resources to complete but allows all other tasks in the address space to continue running. Examples of events that cause delays that the STATUS macro can postpone are:

- Asynchronous exits
- Status stops (by issuing the STATUS macro with the STOP option)
- Timer exits
- Dumping
- Swapping
- Attention exits

Process-must-complete mode also prevents the CANCEL command from stopping a program that is already running. Issuing STATUS SET,MC,PROCESS does not, however, postpone external interrupts or interrupts from I/O.

There are other restrictions to consider when deciding whether to place a process in process-must-complete mode. While in process-must-complete mode, a task cannot:

- Issue STATUS STOP,SRB or STATUS START,SRB.
- Request the LOCAL lock unconditionally
- Issue an SVC or invoke services that issue SVCs
- Issue the WAIT macro or invoke services that issue WAITs.

Remaining in process-must-complete mode for an extended period of time will degrade the performance of other programs waiting to use global resources that the program in this mode holds. Ensure that the process that issues STATUS SET,MC,PROCESS really **must** complete without delay, and that you end process-must-complete mode as soon as the program releases the resource.

To end process-must-compete mode, issue the STATUS macro with the RESET,MC,PROCESS parameter.

STATUS also stops started subtasks and restarts them as described in *z/OS MVS Programming: Assembler Services Guide*.

Communicating with a program (EXTRACT, QEDIT)

The operator can pass information to the started program by issuing a STOP or a MODIFY command. In order to accept these commands, the program must be set up in the following manner.

The program must issue the EXTRACT macro to obtain a pointer to the communications ECB and a pointer to the first command input buffer (CIB) on the CIB chain for the task. The ECB is posted whenever a STOP or a MODIFY command is issued. The EXTRACT macro is written as follows, and returns what is indicated in Figure 1.

```
EXTRACT answer area,FIELDS=COMM
```

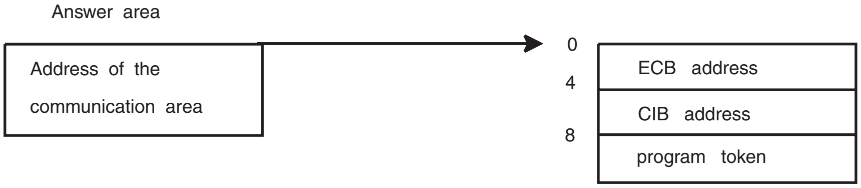


Figure 1. EXTRACT ECB Address, CIB Address, and Program Token

The CIB contains the information specified on the STOP, START, or MODIFY command. If the job was started from the console, the EXTRACT macro will point to the START CIB. If the job was not started from the console, the address of the first CIB will be zero. For more information on the contents of the command input buffer (CIB) and mapping macro IEZCIB, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

If the address of the START CIB is present, use the QEDIT macro to free this CIB after any parameters passed in the START command have been examined. The QEDIT macro is written as follows:

```
QEDIT ORIGIN=address of pointer to CIB,BLOCK=address of CIB
```

Note:

1. The address of the pointer to the CIB is the contents of the answer area plus 4 bytes, as shown in Figure 1.
2. The address of the CIB must be the exact address returned by EXTRACT, not an address generated from copying the CIB to another location.

The CIB counter should then be set to allow CIBs to be chained and MODIFY commands to be accepted for the job. This is also accomplished by using the QEDIT macro:

```
QEDIT ORIGIN=address of pointer to CIB,CIBCTR=n
```

The value of n is any integer value from 0 to 255. If n is set to zero, no MODIFY commands are accepted for the job. However, STOP commands are accepted for the job regardless of the value set for CIBCTR. After a STOP command is issued, the system sets the value of CIBCTR to zero to prevent more modify CIB's from being chained. To continue processing modify's after a STOP command, you must use the QEDIT macro to set the CIBCTR to non-zero.

Note: When using the address or addresses returned from the EXTRACT macro as input to the QEDIT macro, you must establish addressability through the COM data area (mapped by IEZCOM), based on the address returned by the EXTRACT. For a description of the COM data area, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

For the duration of the job, your program can wait on or check the communications ECB at any time to see if a command has been entered for the program. Check the verb code in the CIB to determine whether a STOP or a MODIFY command has been entered. After processing the data in the CIB, issue a QEDIT macro to free the CIB.

The communications ECB is cleared by QEDIT when no more CIBs remain. Care should be taken if multiple subtasks are examining these fields. Any CIBs not freed by the task are unchained by the system when the task is terminated. The area addressed by the pointer obtained by the EXTRACT macro, the communications ECB, and all CIBs are in protected storage and may not be altered.

The portion of the program follows the procedure outlined in the preceding paragraphs. It shows how you can code the EXTRACT and QEDIT macros to accept MODIFY and STOP commands. The full example would use reentrant code.

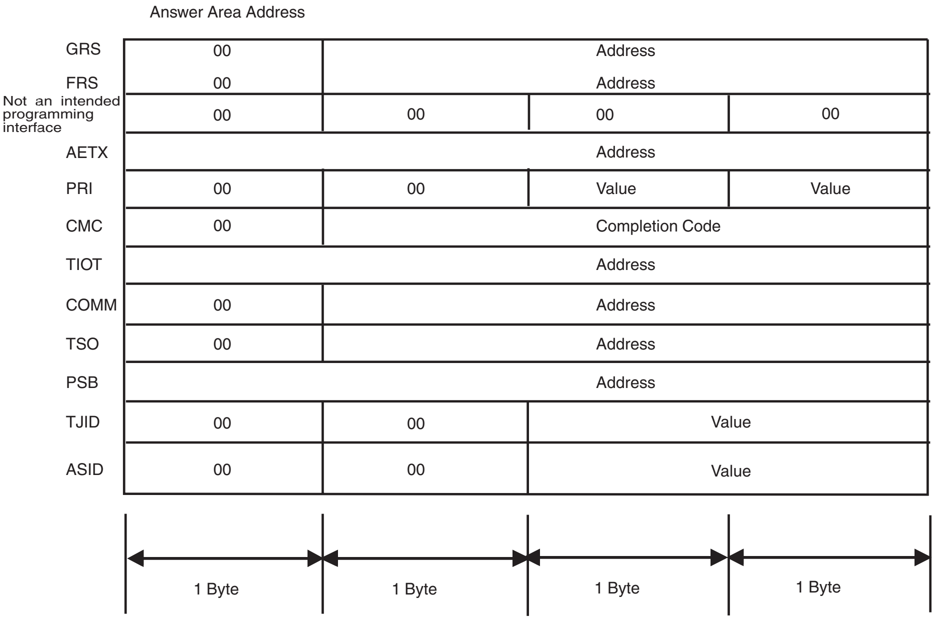
```
R1       EQU   1
R7       EQU   7
R9       EQU   9
R10      EQU   10
R14      EQU   14
         USING *,R10              MODULE ADDRESSABILITY
         LA    R9,COMADDR         GET COMMUNICATIONS AREA
*                                 ADDRESS AT COMADDR
*-------------------------------------------------------------------*
*        OBTAIN ADDRESS OF THE CIB                                  *
*-------------------------------------------------------------------*
         EXTRACT (R9),FIELDS=COMM,MF=(E,EXTRACT)
*                                 EXTRACT THE COMMUNICATIONS AREA
         L     R9,COMADDR         GET ADDRESS OF THE AREA
         USING COM,R9             USE R9 AS BASE ADDRESS OF COMM AREA
         ICM   R7,15,COMCIBPT     GET CIB ADDRESS FROM COM AREA
         BZ    NOCIB              NO CIB, TASK WAS NOT STARTED
         BAL   R14,DOCIB          PROCESS THE CIB
NOCIB    DS    0H
         QEDIT ORIGIN=COMCIBPT,CIBCTR=5   SET MODIFY LIMIT TO 5
         L     R1,COMECBPT        GET ADDRESS OF THE COMMUNICATION ECB
         O     R1,HIBITON         SET HIGH BIT - LAST ECB IN LIST
         ST    R1,MODECB          PUT ADDR OF MODIFY ECB IN LIST
*
*
*
WAIT     DS    0H
         WAIT  ECBLIST=ECBS       WAIT FOR A MODIFY/STOP
*
*  WHEN POSTED HERE, A MODIFY OR STOP HAS BEEN ISSUED
         ICM   R7,15,COMCIBPT     GET CIB ADDRESS FROM COM AREA
         USING CIB,R7             BASE CIB MAPPING
         CLI   CIBVERB,CIBMODFY   WAS IT A MODIFY?
         BNE   NOTMDFY            NO, GO FREE CIB
         BAL   R14,DOCIB          IT WAS A MODIFY, GO PROCESS COMMAND
*
*
*
*------------------------------------------------------------------ *
*        FREE THE CIB                                                *
*------------------------------------------------------------------ *
NOTMDFY  DS    0H
         BAL   R14,DELCIB         FREE CIB
         CLI   CIBVERB,CIBSTOP    WAS IT A STOP?
         BE    EXITRTN            BRANCH TO ROUTINE HANDLING ERRORS
         B     WAIT               WAIT FOR ANOTHER MODIFY
*        .
*        .
DELCIB   DS    0H
*        USE QEDIT TO FREE THE CIB
*        QEDIT WILL ALSO CLEAR THE ECB
```

```
*
         QEDIT ORIGIN=COMCIBPT,BLOCK=(R7)   FREE THE CIB
         BR    R14

DOCIB    DS    0H
*--------------------------------------------------------------------*
* YOUR ROUTINE TO HANDLE CIB PROCESSING WOULD GO HERE.               *
*--------------------------------------------------------------------*
*
*
*
*--------------------------------------------------------------------*
*  CONSTANTS                                                         *
*--------------------------------------------------------------------*
         DS    0F                 FULLWORD ALIGNMENT
HIBITON  DC    X'80000000'        USED TO TURN HIGH ORDER BIT ON
*--------------------------------------------------------------------*
*  FIELDS REQUIRED IN DYNAMIC STORAGE                                *
*--------------------------------------------------------------------*
ECBS     DS    0CL4               ECB LIST FOR WAIT
MODECB   DS    A                  ADDR(MODIFY/STOP ECB)
COMADDR  DS    F                  ADDR(COMAREA) FROM EXTRACT
SV       DS    18F                SAVE AREA
EXTRACT  EXTRACT MF=L             EXTRACT PARAMETER LIST
*--------------------------------------------------------------------*
*  REQUIRED DSECTs                                                   *
*--------------------------------------------------------------------*
COM      DSECT
         IEZCOM   ,                COM AREA
CIB      DSECT
         IEZCIB   ,                CIB
         END
```

Providing an EXTRACT answer area

The EXTRACT macro provides TCB information for either the active task or one of its subtasks. Figure 2 on page 12 shows the order in which the information from the requested fields is returned. If the information from a field is not requested, the associated fullword is omitted. See *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for a description of the EXTRACT answer area fields.



Answer Area Address

GRS	00	Address		
FRS	00	Address		
Not an intended programming interface	00	00	00	00
AETX	Address			
PRI	00	00	Value	Value
CMC	00	Completion Code		
TIOT	Address			
COMM	00	Address		
TSO	00	Address		
PSB	Address			
TJID	00	00	Value	
ASID	00	00	Value	
	1 Byte	1 Byte	1 Byte	1 Byte

Figure 2. EXTRACT Answer Area Fields

You must provide an answer area consisting of contiguous fullwords, one for each of the codes specified in the FIELDS parameter, with the exception of ALL. If ALL is specified, you must provide a 7-word area to accommodate the GRS, FRS, reserved, AETX, PRI, CMC, and TIOT fields. The ALL code does not include the COMM, TSO, PSB, TJID, and ASID fields.

Most of the addresses are returned in the low-order three bytes of the fullword, and the high-order byte is set to zero; the fields for AETX, TIOT, and PSB could have a nonzero first byte. Fields for which no addresses or values are specified in the task control block are set to zero.

For example, if you code FIELDS=(TIOT,GRS,PRI,TSO,PSB,TJID) you must provide a 6-fullword answer area, and the extracted information appears in the same relative order as shown in Figure 2. (That is, GRS is returned in the first word, PRI in the second word, TIOT in the third word, and so forth.)

If FIELDS=(ALL,TSO,PSB,COMM,ASID) is coded, you need an 11-fullword answer area, and the extracted information appears in the answer area in the relative order shown above.

Chapter 3. Program management

You can specify whether you want a program loaded into storage above or below 16 megabytes and whether you want a program loaded at a specific address. This information along with a description of synchronous exits, the use of checkpoint restart, the use of re-entrant modules, refreshing library lookaside facility (LLA) directories, changing the LNKLST concatenation for the system, and specifying modules to be added to or deleted from LPA after IPL is described in this topic.

Load module structures, methods of passing control between programs, and the use of associated macros are described in *z/OS MVS Programming: Assembler Services Guide*,*z/OS MVS Programming: Assembler Services Reference ABE-HSP*. and *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

Residency and addressing mode of programs

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

The addressing modes have the following meanings:

- When a program is running in 24-bit addressing mode, the system treats both instruction and data addresses as 24-bit addresses. This allows programs running in 24-bit addressing mode to address the first 16 megabytes (16,777,216 bytes) of storage in an address space.
- When a program is running in 31-bit addressing mode, the system treats both instructions and data addresses as 31-bit addresses. This allows a program running in 31-bit addressing mode to address the first 2 gigabytes (2,147,483,648 bytes or 128x16 megabytes) of storage.
- The system handles a program in 64-bit addressing mode differently. A program does not get control in 64-bit addressing mode. It starts running in either 24- or 31-bit addressing mode. To change its addressing mode, it uses the SAM64 Assembler instruction. While in 64-bit addressing mode, it continues to run below the 2-gigabyte address of the address space. However, it can access data that exists above the 2-gigabyte address. To give you an idea of the size of the 64-bit address space, consider that a 64-bit address space is 8 billion times the size of a 31-bit address space.

z/OS MVS Programming: Assembler Services Guide provides detailed information concerning the AMODE and RMODE of modules of storage. *z/OS MVS Programming: Extended Addressability Guide* provides more information about how programs use the 64–bit address space.

You can define the residency mode and the addressing mode of a program in the source code. Table 1 on page 14 shows an example of the definition of the AMODE and RMODE attributes in the source code. This example defines the addressing mode of the load module as 31 and the residence mode of the load module as 24. Therefore, the program will receive control in 31-bit addressing mode and will reside below 16 megabytes in 24-bit addressable storage.

Table 1. Assembler Definition of AMODE/RMODE

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

The assembler places the AMODE and RMODE in the output object module for use by the linkage editor. The linkage editor passes this information on to the control program through the directory entry for the partitioned data set that contains the load module. You can also specify the AMODE/RMODE attributes of a load module by using linkage editor control cards. See *z/OS MVS Program Management: User's Guide and Reference* and *z/OS MVS Program Management: Advanced Facilities* for information concerning these control cards.

Placement of modules in storage

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

- 24-specifies that the program must reside in 24-bit addressable storage
- ANY-specifies that the program can reside anywhere in virtual storage

Addressing mode

The AMODE attribute, located in the directory entry for the module, specifies the addressing mode of the module. Bit 32 of the program status word (PSW) indicates the addressing mode of the program that is executing. The system supports programs that execute in 24-bit, 31-bit, or 64-bit addressing mode.

The AMODE attribute can have one of the following values:

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

The 64-bit AMODE attribute works differently. No program receives control in 64-bit addressing mode. To change into 64-bit addressing mode from either 24-bit or 31-bit addressing mode, the program issues the SAM64 Assembler instruction. While in 64-bit addressing mode, the program continues to run below the 2-gigabyte address, but is capable of accessing data above the 2-gigabyte address.

Specifying where the module is to be loaded (LOAD macro)

When a program in supervisor state uses the LOAD macro to bring a copy of the load module into virtual storage, it can use one of three parameters to specify where the control program is to load the module:

- Use the ADDR parameter to load a module in an APF-authorized library at a specified address. You must first allocate storage for the module in your key.
- Use the ADRNAPF parameter to load a module in an unauthorized library at a specified address. You must first allocate storage for the module in your key.
- Use the GLOBAL parameter on LOAD to load the module into either fixed or pageable CSA.
  - GLOBAL=(YES,P) or GLOBAL=YES requests storage in the pageable CSA.

- GLOBAL=(YES,F) requests storage in fixed CSA.

When you use GLOBAL=YES, you can use the EOM parameter to specify when the control program is to delete the module. EOM=NO (the default) requests deletion at task termination. EOM=YES requests deletion at address space termination.

The GLOBAL=YES and GLOBAL=(YES,F) parameters should be used only by programs running in address spaces that never terminate or to load non-executable load modules. A module is loaded into global storage so it can be shared among tasks in different address spaces. If the address space containing the program that loaded the module terminates, the loaded module is deleted and other users of this module could attempt to run a program that is no longer in storage.

Note: If you determine the amount of storage required to hold a loaded module using the module length returned from the BLDL macro in the PDS2 data area field PDS2STOR, this will no longer work for program objects exceeding 16 megabytes. The PDS2STOR field is only 3 bytes in length, and it cannot denote any program length greater than 16 megabytes. You can determine if a program object exceeds 16 megabytes by inspecting the PDS2BIG bit of the PDS2 data area. If the PDS2BIG bit is set on, the program object is greater than 16 megabytes, and the 4-byte PDS2VTSR field contains the amount of storage required to hold a loaded program object greater than 16 megabytes. If the PDS2BIG bit is set off, use the PDS2STOR field to determine the size of storage required.

If you do not use ADDR, ADRNAPF, or GLOBAL=YES (that is, you use GLOBAL=NO or take the default), the control program loads the module in subpool 244 or subpool 251, unless the following three conditions are true:

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

In this case, the control program places the module in subpool 252. When choosing between subpools 244 and 251. the control program uses:

- Subpool 244 only when within a task that was created by ATTACHX with the KEY=NINE parameter
- Subpool 251in all other cases

Subpool 244 is not fetch protected and has a storage key equal to your PSW key. Subpool 251 is fetch protected and has a storage key equal to your PSW key. Subpool 252 is not fetch protected and has storage key 0.

When a program is in problem state, the control program brings the copy of the load module in subpool 251, with one exception. If the module is reentrant, the library is authorized, and you are not running under TSO TEST, the control program places the module in subpool 252.

Synchronous exits (SYNCH or SYNCHX macro)

In general, the SYNCH or SYNCHX macro is used when a program in supervisor state gives temporary control to a processing program routine (not necessarily running in supervisor state) where the processing program is expected to return control to the supervisor state program. The program to which control is given must be in virtual storage when the macro is issued.

When the processing program returns control, the supervisor state bit, the PSW key bits, the system mask bits, and the program mask bits of the program status word are restored to the settings they had before execution of the SYNCH or SYNCHX macro.

When a program is SYNCHed to, it will receive control with the caller's Floating Point Registers and Floating Point Control register. The S/390® linkage convention applies. For more information on the S/390 linkage convention, see "Linkage Conventions" in *z/OS MVS Programming: Assembler Services Guide*.

When the target program is AMODE 24 or AMODE 31, the SYNCH or SYNCHX macro is similar to the BALR in that you can use register 15 for the entry point address. When the target routine is AMODE 64, the SYNCH or SYNCHX macro is similar to LINKX or ATTACHX in that you are expected to use relative branching and register 15 contains a value that can be used to determine the addressing mode of the issuer of the SYNCH or SYNCHX macro as follows:

- Issuer AMODE 24: X'FFFFF000'
- Issuer AMODE 31: X'FFFFF002'
- Issuer AMODE 64: X'FFFFF004'

SYNCH or SYNCHX processing does not save or restore registers when control is returned to the caller unless RESTORE=YES is specified. If you specify RESTORE=NO explicitly or by default, the register contents are unpredictable. When an authorized program uses SYNCH or SYNCHX to invoke an exit in an unauthorized program, the general registers returned from the exit might not contain expected data or correct addresses. Therefore, the authorized program must save the registers in a protected save area and then restore them, or validate the contents of the returned registers, or code RESTORE=YES.

Using checkpoint/restart

When issuing checkpoints and then restarting a task, the restarted task must request control of all resources required to continue processing. Resources are not automatically returned to the task upon restart.

You can use the checkpoint/restart facility with the following restrictions:

- A routine that is restricted from issuing SVCs is also restricted from establishing checkpoints because programmer-designated checkpoints require the use of the checkpoint SVC.
- An exit routine other than the end-of-volume exit routine cannot request a checkpoint.
- A routine invoked by a program call (PC) cannot request checkpoints because the system environment might be different at the time of the restart from what it was at the time of the checkpoint. This could lead to unpredictable results on the return linkage (PT).
- A routine with a PCLINK STACK request outstanding cannot establish a checkpoint.
- Subsystems that use the TCB subsystem affinity service cannot issue checkpoints.
- A program that has selected a data in virtual object using the IDENTIFY service cannot request a checkpoint until it invokes the corresponding UNIDENTIFY service.

For additional information concerning the restrictions and use of the checkpoint/restart facility see *z/OS DFSMSdfp Checkpoint/Restart*.

Using re-entrant modules

When link editing modules as re-entrant, be sure that all the modules and the macros they call are re-entrant. In a multiprocessing system this is important because:

- Two tasks in the same address space, or in different address spaces such as a program in LPA or common storage, making use of the module might cause the module to be executed simultaneously on two different processors.
- Asynchronous appendages can operate on one processor simultaneously with an associated task on another processor.
- Enabled recovery routines can execute on any processor, not necessarily on the one on which the error was detected.

The CSECTs must be unchanged during execution or their critical sections must be explicitly serialized. The general method for ensuring re-entrance of macros is to use the LIST and EXECUTE forms of the macros with a dynamically acquired parameter list.

Using LLACOPY to refresh LLA directories

To enhance performance by reducing I/O overhead, a subsystem (such as CICS®) can keep copies of directories in storage. This allows the subsystem to quickly access modules instead of using I/O to search the directories on DASD.

The LLACOPY macro enables the subsystem to manage the subsystem libraries. It allows the subsystem to keep the library lookaside facility (LLA) directory synchronously updated with the subsystem copy of the data set directory. For information about LLA, see *z/OS MVS Initialization and Tuning Guide*.

LLACOPY synchronously refreshes the LLA directory and returns the new directory entries to the caller. It has the same parameters as the BLDL macro. LLACOPY obtains the requested directory entries from DASD by using BLDL.

If LLACOPY finds the directory entries for the requested names, it fills the caller's BLDL list with those entries. If LLACOPY cannot find a particular member, it does not fill in the entry area in the BLDL list for that member.

If LLACOPY finds the requested names in LLA-managed libraries, it changes the LLA directory by either adding the directory entry (if new) or updating the existing directory entry.

LLACOPY can also inform LLA of members that have been removed from datasets. If LLACOPY does not find the member name in the DASD directory, but the member name is in the LLA directory, LLACOPY removes the LLA directory entry.

LLACOPY returns directory entries to the caller but does not refresh the LLA directory when any of the following instances occur:

- An abend (LLACOPY may not return directory entries).
- The version of MVS installed is prior to Version 3 Release 1.3.
- LLA is not active. There was no LLA directory to update.

- You executed LLACOPY against non-LLA-managed datasets.

For additional information concerning the restrictions and use of LLACOPY, see *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

Changing the LNKLST concatenation

You can use the CSVDYNL macro in an authorized program to change the LNKLST concatenation for associated jobs and address spaces. You can perform the following functions with CSVDYNL:

- Define a LNKLST set by name (REQUEST=DEFINE). A LNKLST set defines the data sets for the LNKLST concatenation.
- Add data sets to the LNKLST set (REQUEST=ADD) or delete data sets from the LNKLST set (REQUEST=DELETE).
- Remove the definition of a LNKLST set (REQUEST=UNDEFINE).
- Test to determine if a module can be located in a LNKLST set (REQUEST=TEST).
- Obtain a list of LNKLST sets and users in the system (REQUEST=LIST).
- Update jobs and address spaces to use the LNKLST set that the system is currently using, called the current LNKLST set (REQUEST=UPDATE).
- Query information about support for LNKLST services (REQUEST=QUERYDYN).

You can perform almost all of the same functions through PROGxx LNKLST statements in SYS1.PARMLIB and dynamically through the SETPROG LNKLST command. For information about PROGxx, see *z/OS MVS Initialization and Tuning Reference*. For information about SETPROG LNKLST, see *z/OS MVS System Commands*.

Changing the current LNKLST set

PROGxx defines the LNKLST sets that can be used by the system. At IPL, you can specify a PROGxx member that activates the current LNKLST set to be used by jobs and address spaces. In PROGxx, you can also specify alternate data sets for SYS1.LINKLIB, SYS1.MIGLIB, SYS1.CSSLIB, SYS1.SIEALNKE, or SYS1.SIEAMIGE to appear at the beginning of the LNKLST concatenation. Unless overridden by these alternates, SYS1.LINKLIB, SYS1.MIGLIB, SYS1.CSSLIB, SYS1.SIEALNKE, and SYS1.SIEAMIGE always occur at the beginning of every LNKLST set as part of the LNKLST concatenation.

You can activate the LNKLST set dynamically through the SET PROG=xx or SETPROG LNKLST ACTIVATE commands. (You cannot use CSVDYNL to activate a LNKLST set.) If the current LNKLST set is dynamically changed, any job or address space associated with the previous LNKLST set continues to use the data sets until the job or address space finishes processing. Thus, a previously current LNKLST set might be active or in use by a job or address space even though a new current LNKLST set has been activated. Jobs or address spaces that are started after the new current LNKLST set is activated use the new current LNKLST set.

For a description and examples of CSVDYNL functions, see *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

Monitoring dynamic LPA processing

Dynamic LPA services allow modules to be added to or deleted from LPA after IPL. Because any product can own LPA modules and have an interest in updating their control structures, an exit (CSVDYLPA) is provided from dynamic LPA services. Providing lists of modules added to or deleted from LPA, the exit is intended to be used as a notification mechanism so that products can update internal control structures with the new module addresses.

The system takes no action as a result of exit routine processing.

CSVDYLPA routine processing. The CSVDYLPA routine, if defined, receives control when the system processes a dynamic LPA services request, either via

- the CSVDYLPA macro,
- the SETPROG LPA operator command, or
- an LPA statement within PROGxx referenced by the SET PROG=xx operator command.

Dynamic LPA services are documented in *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*. The CSVDYLPA exit routine can be used to update control structures that had captured the addresses of particular LPA modules, upon replacement of a module within LPA.

Installing CSVDYLPA. Use the CSVDYNEX macro to connect a routine to the dynamic LPA processing. For example:

```
         CSVDYNEX REQUEST=ADD,                                         X
               EXITNAME=THEEXIT,                                       X
               MODNAME=THEMOD
  .
  .
  .
THEEXIT  DC CL16'CSVDYLPA'
THEMOD   DC CL8'DYLPARTN'
```

The exit routine must be reentrant and AMODE 31.

See “Using dynamic exits services” on page 471 for a description of CSVDYNEX processing.

CSVDYLPA routine environment. The routine receives control in the following environment:

- In supervisor state with PSW key 0.
- In dispatchable unit mode of task, running under the address space, task and request block of the caller of CSVDYLPA.
- In cross memory mode of PASN=HASN=SASN.
- In AMODE 31.
- In primary ASC mode.
- Enabled for I/O and external interrupts.
- With no locks held.
- With parameter areas in the primary address space. The parameter list is described below. The module information area pointed to by the parameter list is mapped by the CSVLPRET macro.

- In the address space in which CSVDYLPA REQUEST=ADD was issued, or the master scheduler address space for the SETPROG or SET PROG operator commands.
- The exit uses fastpath=yes for CSVDYNEX REQUEST=CALL, and will retry (continue calling) on an exit routine error.
- With ENQ resource SYSZCSV.CSVDYLPA held exclusive.
- The programmer should not use CSVDYLPA REQUEST=ADD or CSVDYLPA REQUEST=DELETE within the exit routine.

CSVDYLPA routine recovery. The routine should provide its own recovery, using ESTAE, ESTAEX, or SETFRR EUT=YES. If the exit routine ends abnormally, its recovery routine will get control before the recovery routine established by the CSVDYLPA service.

You can use the ADDABENDNUM and ABENDCONSEC parameters on the CSVDYNEX REQUEST=ADD macro to limit the number of times the exit routine abnormally ends before it becomes inactive. An abend is counted under the following conditions:

- The exit routine does not provide recovery, or the exit routine does provide recovery but percolates the error.
- The system allows a retry, that is, the recovery routine is entered with bit SDWACLUP off.

By default, the system disables the exit routine when it abends on two consecutive calls.

Programming considerations. The programmer should include the CSVLPRET macro which maps one of the input parameters and provides equate symbols for use by the exit routine.

Code the exit routine to be reentrant, and to have AMODE 31.

The exit routine should not modify the parameter areas.

Registers at entry to CSVDYLPA routine. The contents of the registers on entry to a CSVDYLPA routine are:

Register	**Contents**
0	Contains no information for use by the exit routine
1	Address of parameter area
2-12	Contains no information for use by the exit routine
13	Address of 72-byte standard save area
14	Return address
15	Entry point address

Parameter area at entry to CSVDYLPA routine.

Word 1
Address of fullword containing 0 (symbol CsvdylpaFunctionAdd) if processing an ADD request or 1 (symbol CsvdylpaFunctionDelete) if processing a DELETE request.

Word 2
Address of fullword containing the number of module entries in the area pointed to by parameter list word 3.

Word 3
Address of module information entries. The entries are contiguous, each mapped by DSECT LPMEA in macro CSVLPRET. The entries are not sorted. Only entries with bit LpmeaSuccess on should be processed; others should be skipped. When processing a delete request, the information presented for a deleted entry indicates information about the now-current copy of the entry. If, after the deletion, the entry still exists in LPA, the LpmeaEntryPointAddr, LpmeaLoadPointAddr, and LpmeaModlen fields are provided, if available. If no entry still exists in LPA, then a value of X'7FFFFFFF' is presented in the LpmeaEntryPointAddr field, and the LpmeaLoadPointAddr and LpmeaModlen fields do not contain valid data The exit routine must not modify this area, as doing so would compromise the information provided to subsequent exit routines called for the same event.

Word 4
Address of 512-byte work area on a doubleword boundary. This area can be used as a dynamic area by the module, possibly allowing it to avoid doing GETMAIN/FREEMAIN on entry/exit.

Word 5
Address of 12-byte area that is the DDNAME (bytes 0-7) and DCB (bytes 8-11) address used for the ADD request. The DDNAME will be 0s for the following cases:

- a DELETE request;
- an ADD request that specified BYADDR=YES; or
- an ADD request that specified DCB, DCBPTR, MASKDCB, or MASKDCBPTR.

The DCB address will be 0s for the following cases:

- a DELETE request or
- an ADD request that specified BYADDR=YES.

The DDNAME and/or DCB address could be used to locate the name of the data set from which the module was fetched, using field LpmeaSuccessConcatNum to identify the number of the data set within the allocated concatenation.

At the time the exit routine is called, the DDNAME will still be allocated and the DCB will still be open.

Registers at exit from CSVDYLPA routine. Upon return from CSVDYLPA processing, the register contents must be:

Register	**Contents**
0-1	Need not contain any particular value
2-13	Restored to contents at entry
14	Need not contain any particular value
15	0

Disassociating CSVDYLPA. Disassociate the routines from CSVDYLPA when they should no longer receive control. Use the CSVDYNEX macro to disassociate the routines. For example:

```
         CSVDYNEX REQUEST=DELETE,                                      X
               EXITNAME=THEEXIT                                        X
               MODNAME=THEMOD
  .
  .
  .
THEEXIT  DC CL16'CSVDYLPA'
THEMOD   DC CL8'DYLPARTN'
```

Coded example

```
EXITRTN  CSECT
EXITRTN  AMODE 31
EXITRTN  RMODE ANY
*
* entry linkage
*
         STM   R14,R12,12(R13)
         LR    R12,R15
         USING EXITRTN,R12
*
* Processing
*
         L     R2,0(R1)        Get address of function code
         CLC   0(4,R2),FUNCDEL Is this DELETE?
         BE    DONE            Yes, Done
         L     R2,4(R1)        Get address of number of entries
         ICM   R2,15,0(R2)     Get number of entries
         BZ    DONE            No entries, done
         L     R3,8(R1)        Address of module information area
         USING LPMEA,R3        Establish using
TOP      DS    0H              TOP of loop
         TM    LPMEAOUTPUTFLAGS0,LPMEASUCCESS Is this entry valid?
         BZ    NEXT            No, move on to next
         CLC   LPMEANAME,MYMODULE Is this my module?
         BNE   NEXT            No, move on to next
*
* Put code here possibly to move field LpmeaEntryPointAddr to
* your own data area
*
NEXT     DS    0H              Continue loop
         LA    R3,LPMEA_LEN(R3) Point to next entry
         DROP  R3              Release using
         BCT   R2,TOP          Decrement, see if done
DONE     DS    0H
*
* exit linkage
*
         LM    R14,R12,12(R13)
         SLR   R15,R15
         BR    R14
MYMODULE DC    CL8'MODULEA'
FUNCDEL  DC    A(CSVDYLPAFUNCTIONDELETE)
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         CSVLPRET
         END
```

Listing contents of dynamic LPA with CSVDLPAU

You can use `CSVDLPAU` utility to list the modules that comprise the Link Pack Area (LPA). You can choose to display just the modules added by the dynamic LPA facility, or those modules plus the modules that are part of active LPA (for example, MLPA, FLPA, and device support modules) or all of LPA.

Invoking CSVDLPAU

You can invoke CSVDLPAU using the following sample JCL:

```
//jobname  JOB ...
//stepname EXEC PGM=CSVDLPAU,PARM=p
//SYSPRINT DD SYSOUT=s
```

Lower case text reflects the data that you must change.

`EXEC statement parameters:`

Replace p in the sample with one of the following values:

Value	Meaning
D	Display only the dynamic LPA modules (this is the default if no parameter is provided).
A	Display dynamic LPA and active LPA (for example, MLPA, FLPA, and device support) modules.
L	Display all LPA modules (dynamic LPA, active LPA, PLPA).

`DD statement information`

SYSPRINT statement

This DD defines where the output from the program is sent. Replace `s` in the sample with a sysout class that meets your needs and is supported by your installation. Output could instead go to a sequential data set with attributes of `RECFM=FBA, LRECL=133, and BLKSIZE=0`.

`Return codes`

When CSVDLPAU ends, one of the following return codes is placed in general purpose register 15:

Code	Meaning
0	Successful completion.
4	Bad input parameter or could not open SYSPRINT DD.
8	Unexpected error from the CSVINFO service. Message CSVD002W is also written to the SYSPRINT DD file. See “CSVD002W” on page 24.

`Output from CSVDLPAU`

```
CSVD001I title
FLAGS* MODULE    ENTRY PT  LOAD PT   LENGTH
 xfp   modname1  entrypt1  loadpt1   length1
 xfp   modname2  entrypt2  loadpt2   length2
...
* FLAGS EXPLANATION
   D = DYNAMIC LPA
   A = ACTIVE LPA
   L = (P)LPA
   P = ENTIRE MODULE PAGE PROTECTED
   F = (PAGE) FIXED
```

where the values are as follows:

`title` One of the following values:

- DYNAMIC LPA MODULE DISPLAY
- DYNAMIC AND ACTIVE LPA MODULE DISPLAY
- LPA MODULE DISPLAY

`xfp` Identifies three individual flags:

Flag	Meaning
`x`	Indicates one of the following: `D` Dynamic LPA `A` Active LPA `L` PLPA
`f`	Either F to indicate the module is page-fixed or blank to indicate the module is not page-fixed
`p`	Either P to indicate that the entire module is page-protected or blank

`modnameN`
The name of a module

`entryptN`
The entry point address of the module

`loadptN`
The load point address of the module

`lengthN`
The length of the module

CSVD002W CSVINFO WAS NOT SUCCESSFUL. RC=*rc* RSN=*rsn*.

Explanation: The CSVINFO service used by CSVDLPAU did not complete successfully.

In the message text:

rc The hexadecimal return code from CSVINFO.

rsn The hexadecimal reason code from CSVINFO.

Chapter 4. Serialization

Note: In this topic, the term *program* refers to a task or service request block (SRB) routine.

In a multi-tasking, multi-processing environment, resource serialization is needed to coordinate access to resources that are used by more than one program. An example of why resource serialization is needed occurs when one program is reading from a data set and another program needs to write to the data set. To maintain data integrity, the program that needs to write must wait until the program that is reading the data set is finished. Both programs need serialization to coordinate access to the data set.

z/OS provides resource serialization services that programs can use to serialize access to resources. The following examples describe other situations that require serialization:

- Using data in subpools that are shared between programs
- Using data that is referenced by more than one program (for example, attached tasks can execute at the same time as the attaching task on different processors)
- Accessing data sets that are shared among programs in the same address space, if the programs update the data
- Concurrent queue manipulation by programs.

In addition to the serialization function, some of the serialization services provided on MVS can be used to:

- Synchronize the execution of programs that depend on the completion of events in other programs
- Synchronize the execution of programs with minimal overhead.
- Ensure that a routine that owns a critical resource can complete processing of the resource without interruptions that could cause the routine to end.

This information describes the resource serialization services that are available on MVS and the functions that each service performs, and provides guidelines for choosing a serialization service that is appropriate for your application.

Choosing a serialization service

The serialization service that you choose depends on factors such as the type of processing you need to perform and the scope of your application. For example, your application might require that you serialize resources:

- Within a single address space
- Across several address spaces within a single MVS system
- Across multiple MVS systems.

Other requirements for your serialization service might include support for SRB mode callers or cross memory capability.

Table 2 on page 26 shows the resource serialization services that are available on MVS, the functions that each service can perform, and where to find more information about each service:

Table 2. Summary of Serialization Services Available on MVS

Service	Function	Reference
ISGENQ, ENQ/DEQ/RESERVE	• **ISGENQ:** Obtain, change, and release user-defined logical resources (ENQs) and direct access storage device (DASD) Reserves. Serialize resources (such as data sets) within a single MVS address space, single MVS system or across multiple MVS systems. ISGENQ fully supports 64-bit callers.**ENQ/DEQ:** Obtain, change, and release user-defined logical resources (ENQs). IBM recommends using the ISGENQ service over ENQ/DEQ/RESERVE. Serialize resources (such as data sets) within a single MVS address space, single MVS system or across multiple MVS systems. Ensure that a routine can complete its use of a resource without interruptions that could cause the routine to end (must-complete function).**RESERVE:** Allow multiple systems, need not be all MVS, to share direct access storage device (DASD). IBM recommends using the ISGENQ service over ENQ/DEQ/RESERVE. **Note:** Information regarding requests made to the preceding services can be gathered by ISGQUERY or GQSCAN. Also, ISGADMIN can be used to control certain aspects of the preceding services. See ISGADMIN information in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.	• For the ISGENQ macro, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*. For guidance information, see *z/OS MVS Programming: Assembler Services Guide* and *z/OS MVS Planning: Global Resource Serialization*. • For the ENQ and DEQ macros, see *z/OS MVS Programming: Assembler Services Reference ABE-HSP*. For guidance information, see *z/OS MVS Programming: Assembler Services Guide*. For the must-complete function, see "Using the must-complete function (ENQ/DEQ)" on page 34. • "Macros used with shared DASD (RESERVE, EXTRACT, GETDSAB)" on page 37
Latch Manager Callable Services: ISGLCRT, ISGLCR64, ISGLOBT, ISGLOB64, ISGLREL, ISGLRE64, ISGLPB64, ISGLPRG, ISGLPR64, ISGLID, ISGLID 64	Serialize resources (such as control blocks or data sets) within a single address space or across several address spaces in a single MVS system with minimal overhead. Authorized programs only. Synchronize the execution of programs.	"Global resource serialization latch manager" on page 64
Locking (SETLOCK macro)	Serialize system resources (such as MVS system queues or control blocks) within a single MVS system or address space.	"Locking" on page 30
IXLLOCK	Enables authorized applications to obtain shared or exclusive serialization on user-defined logical resources. You can also implement your own locking protocols through the inclusion of user data. IXLLOCK is specific to Sysplex environments and provides high speed cross sysplex sharing. The cost is that it is much harder to use then ENQ and provides less external controls	For guidance information, see *z/OS MVS Programming: Sysplex Services Guide*. For the programming environment, see *z/OS MVS Programming: Sysplex Services Reference*.

If you need to synchronize the execution of programs, Table 3 can help you determine which services to choose. The table compares the attributes of the Pause, Transfer, and Release callable services, the WAIT and POST macros, and the SUSPEND and RESUME macros.

Table 3. Program Serialization Techniques Description

Pause, Release, and Transfer	WAIT and POST	SUSPEND and RESUME
Can change the dispatchability of a task or SRB.	Can change the dispatchability of a task but not an SRB.	Can change the dispatchability of a task or SRB.
Can be called from TCB mode or SRB mode. The interfaces are the same for both modes, and the caller of the services does not need to know whether it is in TCB or SRB mode.	WAIT can only be issued in TCB mode. POST can be issued in TCB or SRB mode. The interfaces used in each mode may be different.	Can be used in both TCB and SRB mode. The invoking program must know if it is in task or SRB mode.
Can release a task or SRB before it is paused	Can post a task before it waits.	Cannot resume either a task or SRB before it is suspended.
An authorized caller can pause and release any task or SRB in the system.	Requires the ECB and task being posted to be in the current primary address space for a non-cross memory post.	No control element is needed.
A task or SRB can only pause on a single Pause Element (PE) at a time.	A task may wait on multiple ECBs. If the wait count numbers are posted, the task is made ready.	Not applicable.
The Transfer service can simultaneously pause one task or SRB and release another.	There is no single service with comparable capability for WAIT and POST.	There is no single service with comparable capability for SUSPEND and RESUME.
The Transfer service can release a task or SRB and immediately pass control to the released task or SRB.	There is no single with comparable capability for WAIT and POST.	There is not single service with comparable capability for SUSPEND and RESUME.
Both authorized and unauthorized callers can use the services.	Both authorized and unauthorized callers can use the macros.	Only authorized callers can use the macros.
The system ensures that pause elements are not reused improperly, thus avoiding improper releases caused by unexpected termination or asynchronous ABENDs.	Callers can incorrectly post a task out of the wrong wait due to post, or even the wrong task, if managed improperly, through task termination.	Callers can resume the wrong task or resume a task from the wrong suspend. Because no control element serializes SUSPENDs and RESUMEs, neither the system nor the user can easily detect any reuse problems.
Local lock is not used by the services; therefore, its usage has no negative effect on local lock contention.	Services using local lock can cause additional contention.	Services may use local lock, and cause some additional contention.
Control the parallel execution of units of work (task or SRB) within the same MVS address space or across MVS address spaces. Allow programs to wait (pause) until one event (RELEASE or TRANSFER) has completed.	Control the parallel execution of tasks. Allow programs to wait until one or more events have completed (event completion function).	Control the parallel execution of request blocks (RBs) and SRBs within a single MVS system or address space.

If you need to serialize resources across multiple MVS systems, you must use the ISGENQ macro or the ENQ/DEQ/RESERVE macro. However, if your application requires resources that reside in a single address space or MVS system, you can choose one of the following services: the CMS lock, ISGENQ, ENQ/DEQ macros, latch manager, the local lock, or IXLLOCK. Table 4 can help you determine which of the serialization services is most appropriate for your application.

Table 4. Comparison of CMS Lock, Local Lock, Latch Manager, and ISGENQ and ENQ/DEQ Serialization Services

Feature	**CMS Lock**	**Local Lock**	**Latch Manager**	**ENQ/DEQ Macros**	**ISGENQ Macro**
Number of items you can serialize individually	One	One	Many	Many	Many
Access allowed	Exclusive	Exclusive	Exclusive and shared	Exclusive and shared	Exclusive and shared
Scope	Single system	Address space	Single system or address space	Address space, single system, or multiple systems	Address space, single system, or multiple systems
Cross memory environment	Any PASN, any HASN, any SASN	Any PASN, any HASN, any SASN	Any PASN, any HASN, any SASN	Any PASN, any HASN, any SASN	Any PASN, any HASN, any SASN.
Unit of work	Task or SRB routine	Task or SRB routine	Task or SRB routine	Task	Task
Minimum authorization	Supervisor state and PSW key 0	Supervisor state and PSW key 0	Supervisor state or PSW key 0-7	Problem state with any PSW key, supervisor state or PSW key 0-7 or APF-authorized for certain types of requests	Problem state with any PSW key, supervisor state or PSW key 0-7 or APF-authorized for certain types of requests
Set-up required	None	None	Application must create one or more latch sets	None	None
Default recovery action	Release owned and pending requests	Release owned and pending requests	Release pending requests, but not owned requests	Release owned and pending requests	Release owned and pending requests

Other techniques, such as those that make use of the compare and swap (CS), compare double and swap (CDS), and test and set (TS) instructions, are provided to perform specialized serialization. For information about those instructions, see *Principles of Operation*.

The capabilities listed in Table 4 are explained below:

Number of items you can serialize individually
The latch manager and ISGENQ allow your application more control over the way it divides resources into individual units and assigns serialization to those resources. This increased control allows you to reduce contention for resources and improve the overall performance of the application. For example, an application can use several latches to serialize resources within an address space, instead of using the address space local lock (of which there is only one) to serialize those resources.

Access allowed
Granting shared access minimizes resource contention because several requestors can read a resource at the same time. The CMS and local locks do not provide shared access to resources. Each unit of work that requests the CMS or the local lock must obtain exclusive access, even though shared access

might be enough. The latch manager and the ISGENQ macro allow programs to request shared or exclusive access to resources.

Scope
The CMS lock and the latch manager can be used to serialize resources only within a single MVS system. The local lock is restricted to a single address space. The ISGENQ macro can be used to serialize resources across multiple MVS systems (as well as a single MVS system).

Cross memory environment
With a cross memory environment established, your application can use the CMS lock, latch manager, or local lock to serialize resources across address spaces within a single MVS system. When the latch manager is being used in a cross memory environment, the current latch set must reside in the primary address space.

Unit of work
The CMS lock, local lock, and latch manager are available to programs that run in task or SRB mode. The ISGENQ macro is available only to programs that run in task mode (SRB routines cannot issue ENQ/DEQ).

Minimum authorization
Programs must run in supervisor state and PSW key 0 to obtain the CMS or local lock, and supervisor state or PKM 0-7 to obtain a latch. The ISGENQ macro is available to programs running in problem state with any PSW key. Supervisor state, PSW key 0-7, or APF authorization is required for certain types of ISGENQ requests.

Set-up required
The CMS lock, local lock, and the ISGENQ macro do not require applications to perform any set-up; the system creates the required structures. Before your application can use the latch manager to serialize resources, the application must create one or more latch sets.

Default recovery action
For the local and CMS locks, the system releases owned and pending serialization requests when the last functional recovery routine (FRR) percolates. For the ISGENQ macro the system releases owned or pending serialization requests during task or address space termination. (See "Releasing the Resource" in *z/OS MVS Programming: Assembler Services Guide* for more discussion about what happens when a task associated with an ENQ terminates before the ENQ is released.)

For performance reasons, and because SRBs might not want to release owned latches when errors occur, the system does not release owned latch requests during task or address space termination. The system does, however, release pending latch requests when the requestor's home address space terminates.

In all cases, the application programmer needs to understand the default recovery action for the service that is used. For more information on how to create a recovery environment, see the information that discusses recovery and termination.

Providing ENQ resource information on DISPLAY GRS command

The ENQ API issuer provides a qname, rname, and scope to identify the resource to serialize; this all that the DISPLAY GRS command can use to present information about the resource. In many cases, it is hard for the operator or systems programmer to understand what these values represent. As such, global resource serialization provides the ISGDGRSRES installation exit to allow the application to add additional information for a given resource on the DISPLAY GRS output. See the 'ISGDGRSRES - Display global resource serialization resource exit' in *z/OS MVS Installation Exits* for more details on how to use the exit.

Note: The additional description information which is gotten by calling the exit may be used in other cases beyond the display GRS command where the information can provide additional diagnostic information.

z/OS Automatic Tape Switching (ATS STAR) provides a good example of the ISGDGRSRES exit's use. ATS STAR obtains an ENQ to serialize access to a tape device. Next, the ENQ uses the ATS STAR QNAME of SYSZATS and an RNAME equal to the device's unique node element descriptor (NED). A NED is not in a human readable format, so ATS STAR uses the ISGDGRSRES exit to add the device type and number to the DISPLAY GRS command output.

In the following DISPLAY output, the field: “ADDITIONAL RESOURCE INFORMATION FROM:” was added as a result of ATS STAR's ISGDGRSRES exit translating the RNAME NED into a readable format.

```
ISG343I 09.13.24 GRS STATUS 539
S=SYSTEMS SYSZATS  NED 003490B40VSSGA05B8000002A9
SYSNAME   JOBNAME         ASID     TCBADDR   EXC/SHR      STATUS
SY1       ALLOC4          0027     006FF448  EXCLUSIVE    OWN
ADDITIONAL RESOURCE INFORMATION FROM:  ALLOCAS IEFAIRSX
349S AT UNIT ADDRESS 05B8 (AUTOSWITCH)
```

Locking

A locking mechanism serializes access to resources. This locking technique is only effective, however, if all programs that depend on a resource use the same locking mechanism. Each type of serially reusable resource is assigned a lock. The system controls a hierarchical locking structure with multiple types of locks to synchronize the use of serially reusable resources. The system also handles all functions related to the locks. These functions include obtaining or releasing locks and checking the status of a particular lock on a processor. Use of these functions is restricted to programs running in supervisor state with a PSW key of 0. This prevents unauthorized problem programs from interfering with the system serialization process.

Categories of locks

There are two categories of locks:

- Global locks -- protect serially reusable resources related to more than one address space.
- Local locks -- protect the resources assigned to a particular address space. When the local lock is held for an address space, the owner of the lock has the right to manipulate the queues and control blocks associated with that address space.

Note: The term CML (cross memory local) lock means the local lock of an address space other than the home address space. LOCAL lock means the local lock of the home address space. When written in lower case, local lock means any local-level lock, either the LOCAL or a CML lock.

The CPU and CMS locks are global locks. These global locks provide system-wide services or use control information in the common area and must serialize across address spaces. The local level locks, on the other hand, do not serialize across address spaces, but serialize functions executing within the address space. Table 5 on page 31 summarizes the characteristics of MVS locks.

Table 5. Summary of Locking Characteristics

lock	global	local	spin	suspend	single	multiple (class)
CPU	X		X			X
CMS	X			X	X	
CML		X		X		X
LOCAL		X		X		X

Note: The CPU lock has no real hierarchy except that once a user obtains it, the user cannot obtain a suspend lock. The CPU lock could be considered a pseudo spin lock. It could also be considered multiple because there is one per processor and any number of requestors can hold it at the same time.

Types of locks

The type of lock determines what happens when a function on one processor in an MP system makes an unconditional request for a lock that is held by another unit of work on another processor. There are two major types of locks: spin and suspend. The CPU lock is in a category by itself but could be considered a pseudo spin lock. Descriptions of these types of locks follow:

- Spin lock -- prevents the requesting function on one processor from doing any work until the lock is freed on another processor. A spin lock causes disablement.
- CPU lock -- provides MVS-recognized (valid) disablement for I/O and external interrupts. MVS-recognized disablement refers to disablement that is obtained through a SETLOCK OBTAIN TYPE=CPU request. While a requestor holds the CPU lock, the requestor is physically disabled for I/O and external interrupts.

  Certain macros require the caller to be disabled for I/O and external interrupts. Unless otherwise specified, this means that the caller must have obtained MVS-recognized disablement. MVS does not guarantee preservation of the interrupt status of programs that explicitly disable for I/O and external interrupts through the STNSM instruction.

  Multiple requests can be made by a unit of work for the CPU lock. The CPU lockword contains the cumulative count of requestors who hold the CPU lock. Obtaining the CPU lock increases the ownership count of the CPU lock by 1; releasing the CPU lock decreases the ownership count by 1.
- Suspend locks -- prevent the requesting program from doing work until the lock is available, but allow the processor to continue doing other work. The requestor is suspended and other work may be dispatched on that processor. Upon release of the lock, the suspended requestor is given control with the lock or is redispatched to retry the lock obtain.
- If the caller issues a SETLOCK macro to obtain a spin lock, the caller must reference only page fixed, fixed, or DREF storage.

Examples of lock types

The CPU lock can be considered a pseudo spin lock. The LOCAL, CML, and CMS locks are suspend locks. Their owners receive control enabled and can be interrupted to run higher priority work. If there is another request for the lock while it is held, the requestor is suspended and other work is dispatched.

NOT Programming Interface information

The local lockword contains the ID of the processor on which its owner is dispatched or an indication that the owner is suspended or interrupted. The CMS lockword contains the ASCB address of the locally locked address space that owns the lock. Special IDs are placed in the local lockword whenever the owner of the local lock is not currently executing on a processor because of an interruption or suspension.

End of NOT Programming Interface information

The CMS lock is an enabled global lock because some functions require significant amounts of time under the lock and could impact the responsiveness of the system. By running these functions enabled under the lock, responsiveness is retained at the expense of some increased contention for the lock.

The CPU lock is a disabled spin lock because the functions that run under the lock are of short duration and cannot tolerate interruptions. The cost in system overhead to perform the status saving necessary to accept interruptions and allow switching would offset the gain in responsiveness.

If a lock is unconditionally requested, the lock is unconditionally obtained. If the lock is conditionally requested, the requestor is given the lock if it is available; if the lock is unavailable, control is returned to the caller without the lock. (See the COND and UNCOND parameters on the SETLOCK macro in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.)

Locking hierarchy

The locks are arranged in a hierarchy to prevent a deadlock between functions on the processor(s).

A function on a processor can request unconditionally only those locks that are higher in the hierarchy than the locks it currently holds, thus preventing deadlocks. The hierarchy is shown in Table 5 on page 31, with the first lock listed being the highest lock. The CPU lock is higher in hierarchical order than any of the suspend locks. Therefore once you obtain the CPU lock, you cannot obtain any suspend lock. The CML and LOCAL locks are equal to each other in the hierarchy.

The locks provided by the system in hierarchical order are:

- CPU (processor lock) -- serializes on the processor level, providing system-recognized (valid) disablement.
- CMS (general cross memory services lock) -- serializes on more than one address space where this serialization is not provided by one or more of the other global locks.
- CML (cross memory local lock) -- serializes resources in an address space other than the home address space.
- Local storage lock (LOCAL) -- serializes functions and storage used by the local supervisor within an address space. There is one lock for each address space.

You must hold a local lock, either CML or LOCAL, when requesting the CMS lock. You cannot release the local lock while holding the cross memory services lock.

CML lock considerations

The cross memory local lock (CML) is provided to allow cross memory services to serialize resources in an address space that might not be the home address space. It has the same attributes as the LOCAL lock. (The LOCAL lock refers only to the home address space pointed to by PSAAOLD.) The owner of a CML lock can be suspended for the same reasons as the owner of the LOCAL lock, such as CMS lock suspension or page fault suspension.

In a multi-tasking environment, it is possible for more than one task or SRB in an address space to obtain a local level lock. For example, task A might own the LOCAL lock of its address space while task B in the same address space owns the CML lock of address space C.

To prevent possible system deadlocks, only one lock at the local level can be held at one time by a unit of work. If a CML lock is requested while owning the LOCAL lock, the requestor will be abended. The same is true if the LOCAL lock is requested while owning a CML lock.

Either a CML lock or the LOCAL lock must be held to request the cross memory services lock (CMS).

The requestor of a CML lock must have authority to access the specified address space prior to the lock request. This is accomplished by setting the primary or secondary address space to that specified on the lock request. The specified address space must be non-swappable prior to the obtain request.

Note: The CML lock of the master scheduler address space cannot be obtained. The master scheduler address space lock can only be obtained as a LOCAL lock.

Obtaining, releasing, and testing locks (SETLOCK)

Use the SETLOCK macro to obtain, release, or test a specified lock or set of locks (using the OBTAIN, RELEASE, and TEST parameters). To use SETLOCK, you must be executing in supervisor state with a PSW key of 0. Users of SETLOCK can also be executing in SRB mode, in cross memory mode, in access register or primary address space control (ASC) mode.

Disabled/enabled state for obtain

When the CPU lock is successfully obtained, control returns to the caller with the processor disabled for I/O and external interruptions.

When a suspend type lock is successfully obtained via an unconditional request, control returns to the caller with the processor enabled for I/O and external interruptions.

If a disabled caller unconditionally requests a suspend type lock that is not immediately available, the caller is abnormally terminated.

Disabled/enabled state for release

When the CPU lock is released, control returns to the caller enabled for I/O and external interruptions if there are no more CPU lock requests outstanding (that is, the CPU lock use count is zero).

When a suspend type lock is released, control returns disabled for I/O and external interruptions if the caller was disabled on entry. Otherwise, control returns to the caller enabled for I/O and external interruptions.

Suspend lock instrumentation data

For the following system suspend locks:

- CMSSMF

- CMSEQDQ
- CMSLATCH
- CMS
- Address space local (and CML) locks

the system maintains lock instrumentation data to help debug scalability issues related to lock contention. Although the CMSSMF, CMSEQDQ, and CMSLATCH locks are not programming interfaces, the system collects lock instrumentation data on those locks to aid debugging performance problem related to those locks.

Lock instrumentation data is defined in mapping macro IHALOCKI. Lock instrumentation data is kept by the lock owner and the home address space of the lock requestor when it is different than the lock owner. All suspend locks noted above have a Common Lock Instrumentation Block (LockInst_Comm) and could point to a Unique Lock Instrumentation Block (LockInst_Uniq). Generally, the following set of lock instrumentation data is kept for each lock:

- Number of suspends
- Number of suspends when there was some other unit of work already suspended on the lock requested. This is a subset of the previous count and can be used to get an idea of the frequency multiple work units are suspended for the lock.
- Time suspended.

CMS locks are owned by the system so the lock instrumentation data is kept at the system level. The system Common Lock Instrumentation Block for each CMS lock is pointed to by the ECVT (mapped by IHAECVT) which represents the total CMS lock contention seen on the system. CMS Lock instrumentation data is also kept in the home address space of the work unit that requested the CMS lock. Each address space has a Common Lock Instrumentation Block pointed to by the ASSB (mapped by IHAASSB) for each CMS lock. These blocks represents the contention encountered on CMS locks for units of work from a given address space.

Address space local (and CML) lock instrumentation data is associated with an address space. The Common Lock Instrumentation block for the local lock is pointed to by the ASSB (mapped by IHAASSB). The common block contains lock instrumentation data for suspends on the local lock by units of work in that address space.

Each address space's local lock can be requested from another address space as a CML lock. CML lock instrumentation is kept in a Unique Lock Instrumentation Block called LockInst_Uniq_CML. When a CML lock is requested, two address spaces are involved, so each LockInst_Uniq_CML contains two sets of lock instrumentation data; one for the owner and one for the requestor. For CML lock contention, lock instrumentation data will be updated in the LockInst_Uniq_CML block that owns the CML lock and in the LockInst_Uniq_CML block of the home address space that requested the CML lock.

Using the must-complete function (ENQ/DEQ)

Routines operating under a storage protection key of zero often update and/or manipulate system resources such as system data sets, control blocks, and queues. These resources contain information critical to continued operation of the system. The task requesting this serialization must successfully complete its processing of the resource. Otherwise, the resource might be left incomplete or might contain erroneous information.

The ENQ service routine ensures that a routine queued on a critical resource(s) can complete processing of the resource(s) without interruptions leading to termination. ENQ places other tasks in a non-dispatchable state until the requesting task -- the task issuing an ENQ macro with the set must-complete (SMC) parameter -- has completed its operations on the resource. The requesting task releases the resource and terminates the must-complete condition by issuing a DEQ macro with the reset must-complete (RMC) parameter.

Because the must-complete function serializes operations to some extent, its use should be minimized -- use the function only in a routine that processes system data whose validity must be ensured. Just as the ISGENQ or the ENQ function serializes use of a resource requested by many different tasks, the must-complete function serializes execution of tasks.

Characteristics of the must-complete function

The must-complete function can be used only at the step level, where only the current task in an address space is allowed to execute. All other tasks are made non-dispatchable.

When the must-complete function is requested, the requesting task is marked in “must complete mode” when the resource(s) queued upon are available. All asynchronous exits from the requesting task are deferred. Tasks external to the requesting task are prevented from initiating procedures that will cause termination of the requesting task. Other external events, such as a CANCEL command issued by an operator, or a job step time expiration, are also prevented from terminating the requesting task.

The failure of a task that owns a must-complete resource results in the abnormal termination of the entire job step. The programmer and the operator receive a message stating that the failure occurred while the step was in must-complete mode.

Programming notes

1. All data used by a routine that is to operate in the must-complete mode should be checked for validity to ensure against a program-check interruption.
2. If a routine that is already in the must-complete mode calls another routine, the called routine also operates in the must-complete mode. An internal count is maintained of the number of SMC requests; an equivalent number of RMC requests is required to reset the must-complete function.
3. Interlock conditions can arise with the use of the ENQ function. Additionally, an interlock might occur if a routine issues an ENQ macro while in the must complete mode. Also, a task that is non-dispatchable, because of a must-complete request, might already be queued on the requested resource. In this case, an enabled wait occurs. An enabled wait can be broken by an operator's action (such as the use of the FORCE command).
4. There are some considerations to be aware of when using enclaves for tasks that serialize resources using the ENQ macro. For details, see “Using ENQ/DEQ or Latch Manager Services With Enclaves” in *z/OS MVS Programming: Workload Management Services*.
5. The macros ATTACH, ATTACHX, LINK, LINKX, LOAD, XCTL, and XCTLX should not be used (unless extreme care is taken) by a routine operating in the must-complete mode. An interlock condition results if a serially reusable routine requested by one of these macros either has been requested by one of

the tasks made non-dispatchable by the use of the SMC parameter or was requested by another task and has been only partially fetched.

6. The time a routine is in the must-complete mode should be kept as short as possible -- enter at the last moment and leave as soon as possible. One suggested way is to:
   a. ENQ (on desired resource(s))
   b. ENQ (on same resource(s)),RET=HAVE,SMC=STEP

   Step (a) gets the resource(s) without putting the routine into the must-complete mode. Later, when appropriate, issue the ENQ with the must-complete request (Step b). Issue a DEQ macro to terminate the must complete mode as soon as processing is finished. Tasks set non-dispatchable by the corresponding ENQ macro are made dispatchable and asynchronous exits from the requesting task are enabled.

Shared direct access storage devices (shared DASD)

The shared DASD facility allows systems to share direct access storage devices. Systems can share common data and consolidate data when necessary. No change to existing records, data sets, or volumes is necessary to use the facility. However, the installation must write its applications to share DASD, and the reorganization of volumes might be desirable to achieve better performance.

Exercise careful planning in accessing shared data sets or shared data areas. Data integrity can not be assured without proper intersystem communication. For more information about how shared DASD relates to macros, see “Macros used with shared DASD (RESERVE, EXTRACT, GETDSAB)” on page 37. Similarly, appropriate security procedures must be performed on each of the multiple systems involved in the sharing of DASD before data can be regarded as secure. Data sets that are intended to be protected via passwords or RACF® should be initially protected on each system before sensitive data is placed in them. This topic, as it refers to password protection, is discussed further under “System Configuration”.

Volume/device status

The shared DASD facility requires that certain combinations of volume characteristics and device status be in effect for shared volumes or devices. Table 6 shows the combinations that must be in effect for a volume or device:

Table 6. Valid Volume Characteristic and Device Status Combinations

System A	Systems B, C, D
Permanently resident	Permanently resident
Reserved	Reserved
Removable	Offline - Non-JES3 devices
Removable	Removable - JES3 - managed devices
Offline	Removable, reserved, or permanently resident (In JES2, if a device is removable in one system, it must be offline in all others.)

If a volume or device is marked removable on any one system, the device must be either in offline status or removable status on all other systems. The mount characteristic of a volume and/or the status of a device can be changed on one system as long as the resulting combination is valid for other systems sharing the volume or device. No other combinations of volume characteristics and device status are supported.

Volume handling

Volume handling with the shared DASD option must be clearly defined because operator actions on the sharing system must be performed in parallel. The following rules should be in effect when using the shared DASD option:

- Operators should initiate all shared volume mounting and demounting operations. The system will dynamically allocate devices unless they are in reserved or permanently resident status, and only the former can be changed by the operator.
- Mounting and demounting operations must be done in parallel on all sharing systems. A VARY OFFLINE must be issued on all systems before a device can be demounted.
- Valid combinations of volume mount characteristics and device status for all sharing systems must be maintained. To IPL a system, a valid combination must be established before device allocation can proceed. This valid combination is established either by specifying mount characteristics of shared devices in VATLST, or varying all devices that can be shared offline before issuing START commands and then following parallel mount procedures.

Macros used with shared DASD (RESERVE, EXTRACT, GETDSAB)

When using shared DASD, IBM recommends that your application issue ISGENQ SCOPE=SYSTEMS RESERVEVOLUME=YES or ENQ SCOPE=SYSTEMS and define the resource name on which you are serializing, rather than use the RESERVE macro. However, to ensure successful serialization of shared DASD, you can use RESERVE if one of the following is true:

- Global resource serialization is not active
  - ISGENQ services are not available until the Global Resource Serialization address space is initialized. This is very unlikely and only true if your code runs very early in the IPL processing (prior to the Global Resource Serialization address space being initialized). The Global Resource Serialization address space is created and available in all Global Resource Serialization modes and therefore all z/OS environments.
- Your installation is not using SMS to manage the DASD.
  - The RESERVE application programming interfaces, either through the RESERVE or ISGENQ macros, can result in a DASD RESERVE or an ENQ being obtained, or both. In all cases, an ENQ is obtained that is defined by the caller through the QNAME/RNAME. The scope of the ENQ and whether the hardware RESERVE is issued by the system is controlled by several factors:
    - A hardware RESERVE is not issued when one of the following is true:
      1. The volume is not defined as being SHARED through the IODF.
      2. The volume is mapped by Virtual I/O (VIO).
      3. The current Global Resource Serialization RNL indicates that the RESERVE should be converted to a SYSTEMS level ENQ (The resource does not match an entry in the SYSTEMS EXCLUSION RNL and does match an entry in the RESERVE Conversion RNL)
      4. The RESERVE is converted by the ISGNQXITBATCH exit, or ISGNQXITBATCHCND exit, or both.
    - The scope of the ENQ originates as SYSTEMS:
      1. If the above volume attributes indicate that the system should issue the hardware RESERVE, the associated ENQ can be changed to SYSTEM

(local ENQ) when the QNAME/RNAME match an entry in the exclusion RNL. In general, this would be the case when the installation wants the RESERVE to occur and therefore does not require the associated SYSTEMS ENQ. The SYSTEM ENQ in this case would serialize the device across multiple users on the same system. The RESERVE serialized the device across systems.

2. The ENQ can stay as a SYSTEMS ENQ in cases where the RESERVE is not issued due to the conversion RNL. In this case, the ENQ serializes the resource across z/OS systems.

All of these factors are generally not a concern of the caller of the RESERVE service. The caller is using RESERVE because it supports serialization of the volume across systems through hardware reserve. The installation controls the factors defined above based on its requirements.

You can also use the ENQ, DEQ, EXTRACT, and GETDSAB macros when working with shared DASD.

To obtain the address of a task input/output table (TIOT) entry, from which you can obtain the unit control block (UCB) address of a device, you can use the GETDSAB macro or the EXTRACT macro. Your installation may have installed products that require the use of the GETDSAB macro to obtain the address of the product's TIOT entries. If you plan to use the EXTRACT macro, first check the documentation for the related product to ensure that the product does not require the use of the GETDSAB macro.

- The RESERVE macro reserves a device for use by a particular system. You identify the device you want reserved by its symbolic resource name and UCB address. For a 31-bit address, you must specify the LOC=ANY parameter. Each task that needs exclusive use of a device must issue the RESERVE macro. When RESERVE is in effect for a device, MVS first ensures that the device is reserved for the appropriate system.

  Note:

  1. The set-must-complete (SMC) parameter of the ENQ macro can also be used with RESERVE.
  2. If a checkpoint restart occurs when a RESERVE is in effect for devices, the system does not restore the RESERVE; the user's program must reissue the RESERVE.
- The SYSDSN major name must be in the GRSRNL inclusion list. See *z/OS MVS Planning: Global Resource Serialization* for information about GRSRNL and dynamically changing GRSRNL. See *z/OS MVS Initialization and Tuning Reference* for information about the GRSRNL parmlib member.
- When the task issues a DEQ for the resource named on the RESERVE macro, the system reduces the count in the UCB. When this count reaches zero, the system ensures that the device is unreserved.
- If global resource serialization is active, ISGENQ, ENQ, and DEQ with SCOPE=SYSTEMS specified, can serialize on a particular shared DASD data set without reserving the entire device. See *z/OS MVS Planning: Global Resource Serialization* for details.
- The EXTRACT macro returns the address of the task input/output table (TIOT) from which the UCB address can be obtained. “Finding the UCB address for the RESERVE macro” on page 40 explains this procedure. EXTRACT provides information; it does not actually serialize a resource.

- The GETDSAB macro returns the address of the data set association block (DSAB) associated with a DD name. The DSAB contains the address of the task input/output table (TIOT) entry for the DD name, from which the UCB address can be obtained. GETDSAB provides information; it does not actually serialize a resource.

  The macro has been enhanced to include a LOC parameter. This controls whether the search for the address of the DSAB should be restricted to those which reside below the 16Mb line (LOC=BELOW) or should examine both above and below the line DSABs (LOC=ANY). The default is to restrict the search to below the 16 Mb line.

Releasing devices

The DEQ macro is used with RESERVE just as it is used with ENQ. It must describe the same resource as the RESERVE and its scope must be stated as SYSTEMS. However, the UCB= parameter is only allowed if the DEQ macro is issued by a task that has previously reserved a device. If the DEQ macro is not issued by a task that has previously reserved a device, the system frees the device when the task is terminated. For a 31-bit UCB address, you must specify the LOC=ANY parameter.

Preventing interlocks

The greater the number of device reservations occurring in each sharing system, the greater the chance of interlocks occurring. Allowing each task to reserve only one device minimizes the exposure to interlock. The system cannot detect interlocks caused by a program's use of the RESERVE macro and therefore, enabled wait states can occur on the system. Global resource serialization can also be used to prevent interlocks by suppressing the hardware RESERVE or simply issuing a global ISGENQ to serialize the resource. See *z/OS MVS Planning: Global Resource Serialization* for additional information on this topic.

Timing contention

You can set a timer with ECB and also use the ECB option on ISGENQ, ENQ, and RESERVE services to measure the amount of time you want to wait for the resource to become available. First, you wait on both ECBs. When one ECB is posted, check if the posted ECB was for the timer or for the GRS service. If the timer expired first, use DEQ or ISGENQ REQUEST=RELEASE to release the GRS resource. For an ENQ resource, before you use DEQ to release the resource, you might consider using the ISGECA service to determine the possible blocker or blockers of the resource. The amount of time that you wan to wait depends on the tolerance of your application. The time cannot be determined in some cases. See *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* and *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU* for more information about the ECB option on ISGENQ, ENQ and RESERVE services.

Volume assignment

Because exclusive control is by device, not by data set, consider which data sets reside on the same volume. In this environment it is quite possible for two tasks in two different systems -- processing four different data sets on two shared volumes -- to become interlocked. (If global resource serialization is active and RESERVEs are converted to global ENQs, an interlock does not occur.) For example, as shown in Figure 3 on page 40, data sets A and B reside on device 124, and data sets D and E reside on device 236. A task in system 1 reserves device 124 in order to use data set A; a task in system 2 reserves device 236 in order to use data set D. Now the task in system 1 tries to reserve device 236 in order to use data set E and the task in system 2 tries to reserve device 124 in order to use data set B. Neither can ever regain control, and neither will complete normally. When the system has job step time limits, the task, or tasks, in the interlock will be abnormally terminated when the time limit expires. Moreover, an interlock could mushroom, encompassing new tasks as these tasks try to reserve the devices involved in the existing interlock.

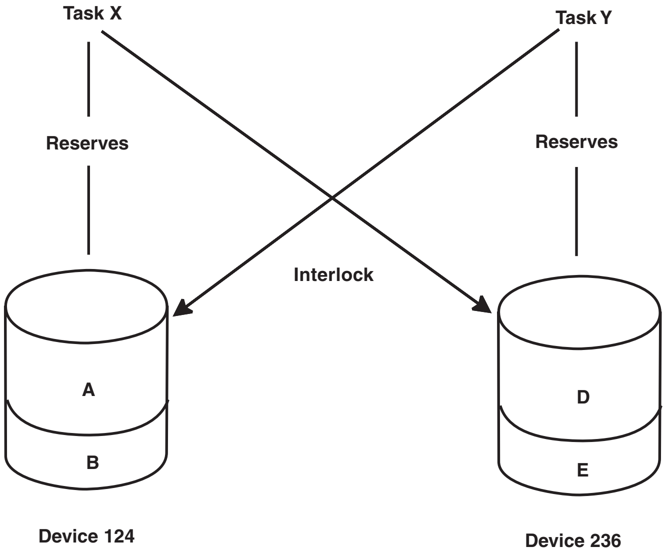


Figure 3. Example of an Interlock Environment

Program libraries

When assigning program libraries to shared volumes, take care to avoid interlock. For example, SYS1.LINKLIB for system 1 resides on volume X, while SYS1.LINKLIB for system 2 resides on volume Y. A task in system 1 invokes a direct access device space management function for volume Y, causing that device to be reserved. A task in system 2 invokes a similar function for volume X, reserving that device. However, each load module transfers to another load module via XCTL. Because the SYS1.LINKLIB for each system resides on a volume reserved by the other system, the XCTL macro cannot complete the operation. An interlock occurs; because no access to SYS1.LINKLIB is possible, both systems will eventually enter an enabled wait state. (If global resource serialization is active and RESERVEs are converted to global ENQs, an interlock does not occur.)

Using different serialization techniques for the same volume

A task interlock can occur within a global resource serialization complex when two tasks reserve the same volume and some of the RESERVEs specify resource names that suppress the hardware reserve while other RESERVEs are hardware reserves that lock up the entire volume. The UCB count of outstanding reserves for that volume is manipulated only for the hardware RESERVEs.

If you code a RESERVE macro, the hardware reserve is suppressed when the resource name appears in the reserve conversion resource name list (RNL). See *z/OS MVS Planning: Global Resource Serialization* for additional information about RNLs and about preventing interlocks.

Finding the UCB address for the RESERVE macro

When using shared DASD, IBM recommends that your applications issue ENQ SCOPE=SYSTEMS and define the resource name on which you are serializing, rather than using the RESERVE macro. However, to ensure successful serialization of shared DASD, you need to use RESERVE if one of the following is true:

- Global resource serialization is not active

- Your installation is not using SMS to manage the DASD.

This topic explains procedures for finding a particular UCB among those that are allocated to the current address space for use by the RESERVE macro. To search other UCBs, you can issue certain other macros but it is unreliable unless you serialize those accesses. While your data sets remain allocated, the UCBs will not disappear. It also shows a sample assembler language subroutine that issues the RESERVE and DEQ macros and can be called by routines written in higher level languages.

Using the GETDSAB macro to find the UCB address: This procedure can be used by application programs that are the only task running in an address space, for either non-concatenated DD statements or data sets that reside on a single volume.

To find the UCB address for a DD name without an Open DCB, you can provide a DD name, an open DCB or ACB or a known data set association block (DSAB) address or request the first DSAB address. From this address, calculate the address of the associated TIOT entry, and the corresponding UCB address.

Procedures: For any allocated data set:

1. Invoke the GETDSAB macro, specifying the name of the desired DD. The system returns the pointer to the DSAB associated with the required TIOT entry. This procedure works only for allocations that did not use the XTIOT or UCB nocapture options of dynamic allocation.
2. Use the IHADSAB macro to map the DSAB. Get the word in DSABTIOT. It is at offset 16. It is a pointer to the TIOT entry.
3. Use the IEFTIOT1 macro to map the TIOT entry, which begins at TIOELNGH. The first UCB address is the 3-byte field at TIOEFSRT. It is at offset 17.
4. Issue the RESERVE macro, specifying the address obtained in step 3 as the parameter of the UCB keyword. Do not code the LOC=ANY option because this UCB address is 24-bit.

Using the IEFDDSRV macro to find the UCB address: This procedure can be used for any DD statement.

To get a list of UCB addresses for the DD name, invoke the IEFDDSRV macro.

Procedures: For any allocated data set:

1. Invoke the IEFDDSRV macro. You can specify the name of the desired DD, the address of an open DCB or ACB or the address of the DSAB. If you want to support the XTIOT or UCB nocapture option of dynamic allocation (which is recommended), also code LOC=ANY on the macro call. This results in returning a list of the UCB addresses for the DD.
2. Issue the RESERVE macro, specifying one of the addresses obtained in step 1 as the parameter of the UCB keyword.

Using the EXTRACT macro to find the UCB address: This procedure can only be used by application programs that are the only task running in an address space, for either non-concatenated DD statements or data sets that reside on a single volume. This procedure works only for allocations that did not use the XTIOT or UCB nocapture options of dynamic allocation.

Use the EXTRACT macro to obtain information from the task control block (TCB). The address of the TIOT can be obtained from the TCB in response to an

EXTRACT macro. Before issuing an EXTRACT macro, set up an answer area to receive the requested information. One fullword is required for each item to be provided by the control program. If you want to obtain the TIOT address, you must specify FIELDS=TIOT in the EXTRACT macro.

The control program returns the address of the TIOT, right adjusted, in the fullword answer area.

Procedures:

1. Issue the EXTRACT macro with the FIELDS=TIOT parameter to extract the TIOT from the TCB.
2. Search the TIOT for the DD name associated with the shared data set.
3. When you find the entry for the required DD name, obtain the UCB address from field TIOEFSRT.
4. Issue the RESERVE macro specifying the address obtained in step 3 as the parameter of the UCB keyword.

Using the DEB and the DCB to find the UCB address: You can also obtain the UCB address via the data extent block (DEB) and the data control block (DCB). The DCB contains data pertinent to the current use of the data set. After the DCB has been opened, field DCBDEBA contains the DEB address. The DEB contains an extension of the information in the DCB. Each DEB is associated with a DCB and the two point to each other.

The DEB contains information about the physical characteristics of the data set and other information that the control program uses. A device-dependent section for each extent is included as part of the DEB. Each such extent entry contains the UCB address of the device to which that portion of the data set has been allocated. In order to find the UCB address, you must locate the extent entry in the DEB for which you intend to issue the RESERVE macro. (In disk addresses of the form MBBCCHHR, the M indicates the extent number starting with 0.)

Procedures:

- For opened data sets:
  1. Load the DEB address from the DCB field labeled DCBDEBA.
  2. The DEB is mapped by the IEZDEB macro. Load the address of the field labeled DEBUCBA. The result is a pointer to the UCB address in the DEB.
  3. Test the DEB31UCB bit before issuing the RESERVE macro with the address of the word obtained in step 2 as the parameter of the UCB keyword. If that bit is zero, issue RESERVE without LOC=ANY. This informs RESERVE that the UCB address is 24-bit. If DEB31UCB is one, code LOC=ANY on the RESERVE macro because the UCB address is four bytes and might point above the line.
- For BDAM data sets, you can reserve the device at any point in the processing in the following manner:
  1. Open the data set.
  2. Convert the block address used in the READ/WRITE macro to an actual device address of the form MBBCCHHR.
  3. Load the DEB address from the DCB field labeled DCBDEBA.
  4. Load the address of the field labeled DEBBASND in the DEB.
  5. Shift the number 1 left by the value in the DEBEXSCL byte. It is the extent scale and is the power of two that indicates the length of each extent description. For DASD, the extent scale currently is 4.

6. Multiply the “M” of the actual device address (step 2) by the result of the previous step that currently is 16.
7. The sum of steps 4 and 5 is the address of the correct extent entry in the DEB for the next READ/WRITE operation. The sum is also a pointer to the UCB address for this extent.
8. Issue the RESERVE macro specifying the address obtained in step 6 as the parameter of the UCB keyword.

- For information concerning how to find the UCB address when using the VSAM access method, see *z/OS DFSMS Using Data Sets*.

RESDEQ subroutine using IEFDDSRV: The assembler language subroutine in Figure 4 on page 44 can be used by assembler language programs to issue the RESERVE and DEQ macros. Parameters that must be passed to the RESDEQ routine, if the RESERVE macro is to be issued, are:

DDNAME - the eight character name of the DD statement for the device to be reserved.

QNAME - an eight character name.

RNAME LENGTH - one byte (a binary integer) that contains the RNAME length value.

RNAME - a name from 1 to 255 characters in length.

The DEQ macro does not require the UCB=*ucb addr* as a parameter. If the DEQ macro is to be issued, a fullword of binary zeroes must be placed in the leftmost four bytes of the DDNAME field before control is passed.

```
 RESDEQ   CSECT
 RESDEQ   AMODE 24
 RESDEQ   RMODE 24
          SAVE  (14,12),T        SAVE REGISTERS
          BALR  12,0             SET UP ADDRESSABILITY
          USING *,12
          ST    13,SAVE+4
          LA    11,SAVE          ADDRESS OF MY SAVE AREA IS
 *                               STORED IN THIRD WORD OF CALLER'S
          ST    11,8(13)         SAVE AREA
          LR    13,11            ADDRESS OF MY SAVE AREA
          LR    9,1              ADDRESS OF PARAMETER LIST
          L     3,0(9)           DDNAME PARAMETER OR WORD OF ZEROES
          CLC   0(4,3),=F'0'     WORD OF ZEROS IF DEQ IS REQUESTED
          BE    WANTDEQ
 *PROCESS FOR DETERMINING THE UCB ADDRESS USING IEFDDSRV.
          IEFDDSRV RETRIEVE,DEVENTRY,DDNAME=(3),LOC=ANY,          X
              DEVAREA=DDINFO   INVOKE IEFDDSRV USING DDNAME
          LTR   15,15            CHECKS RETURN CODE
          BZ    FINDUCB          BRANCH IF RETURN CODE IS ZERO
          ABEND 200,DUMP         DDNAME IS NOT IN TIOT, ERROR
 FINDUCB  L     11,DDINFO        ADDRESS OF DD INFORMATION
                USING DVAR,11
          L     8,DVAR_DEVLST_ADDR  ADDRESS OF AN ENTRY
          USING DVAR_DEVICE_LIST,8
*PROCESS FOR DETERMINING THE QNAME REQUESTED
 WANTDEQ  L     7,4(9)           ADDRESS OF QNAME
          MVC   QNAME(8),0(7)    MOVE IN QNAME
 *PROCESS FOR DETERMINING THE RNAME AND THE LENGTH OF RNAME
          L     7,8(9)           ADDRESS OF RNAME LENGTH
          MVC   RNLEN+3(1),0(7)  MOVE BYTE CONTAINING LENGTH
          L     7,RNLEN
          STC   7,RNAME          STORE LENGTH OF RNAME IN THE
 *                               FIRST BYTE OF RNAME PARAMETER
 *                               FOR RESERVE/DEQ MACROS
          L     6,12(9)          ADDRESS OF RNAME REQUESTED
          BCTR  7,0              SUBTRACT ONE FROM RNAME LENGTH
          EX    7,MOVERNAM       MOVE IN RNAME
          CLC   0(4,3),=F'0'
          BE    ISSUEDEQ
          RESERVE (QNAME,RNAME,E,0,SYSTEMS),UCB=DVAR_DEV_ADDR
          * FREE THE AREA RETURNRF BY IEFDDSRV.
          SR    0,0              PREPARE FOR IC
          IC    0,B'0111',DVAR_LENGTH GET AREA LENGTH
          SR    8,8
          DROP  8
          IC    8,DVAR_SUBPOOL   GET SUBPOOL
          STORAGE RELEASE,LENGTH=(0),ADDR=(11),SUBPOOL=(8)
          DROP  11
          B     RETURN
 ISSUEDEQ DEQ   (QNAME,RNAME,0,SYSTEMS)
 RETURN   L     13,SAVE+4        RESTORE REGISTERS AND RETURN
          RETURN (14,12),T
 MOVERNAM MVC   RNAME+1(0),0(6)
 SAVE     DS    18F              STANDARD SAVE AREA
 DDINFO   DS    A                ADDRESS OF DD INFORMATION
 QNAME    DS    2F               MAJOR NAME FOR RESERVE AND DEQ MACROS
 RNAME    DS    CL256            MINOR NAME
 RNLEN    DC    F'0'             LENGTH OF MINOR NAME
          END
```

Figure 4. Example of Subroutine Issuing RESERVE and DEQ Using GETDSAB

Serializing parallel tasks (WAIT and POST)

Use WAIT and POST to control the parallel execution of tasks as a means of serialization. The basic WAIT/POST services that are available to both authorized and unauthorized programs are explained in the *z/OS MVS Programming: Assembler Services Guide*. The additional forms of WAIT and POST that are available only to authorized programs are explained in this topic. The additional forms are:

- Cross memory post (ASCB parameter specified).
- Directly posting an ECB without invoking POST.
- A branch entry for the EVENTS service.
- POST exit routine defined by extended ECBs.
- A branch entry for the POST service.
- A branch entry for the WAIT service.

Asynchronous cross memory POST

The POST macro signifies the completion of an event by one routine to another. Usually the system posts the completion of the event in the user's address space. The user can, however, cause the system to post completion of the event in another address space.

The authorized user (executing in supervisor state, under protection key 0-7, or APF-authorized) of the POST macro can use the ASCB and ERRET parameters to perform a POST in an address space other than the user's own. If the caller is authorized to specify the ASCB and ERRET parameters, no check is made to determine if the requested address space is the issuing address space. This use of the POST macro is sometimes known as “asynchronous cross memory post”.

Note: Do not issue asynchronous cross memory Post more frequently than necessary. Large numbers of cross memory Posts can consume excessive amounts of system resources.

The ERRET routine is given control when POST detects an error after control has been returned to the issuer of POST. If the ERRET routine does not functionally need to do anything, use CVTBRET or CVTRCZRT in the CVT data area as the ERRET routine.

When LINKAGE=SYSTEM is not specified

If MEMREL=YES is coded (or defaulted) on the POST macro, the ERRET routine is given control in the issuer's home address space. If MEMREL=NO is coded on the POST macro, the ERRET routine is given control in ASID 1.

When LINKAGE=SYSTEM is specified

The ERRET routine is given control in the issuer's primary address space.

The routine executes in supervisor state and PSW key 0. It receives control enabled, unlocked, in SRB mode, and with the following register contents:

Register	**Contents**
0	ECB address
1	ASCB address (address specified for ASCB keyword)
2	completion code specified on POST invocation
3	system completion code that indicates why the POST request failed

4-13 used as a work register by the system

14 return address

15 ERRET address

The ERRET routine will receive control in the addressing mode of the caller of the cross memory POST, and the ERRET routine will run in a cross memory environment where home, primary, and secondary address spaces are all equal. The ERRET routine must return control to the address in register 14, unlocked and enabled.

If cross-memory post is being used, a synchronization problem arises when it becomes necessary to eliminate an ECB that is a potential target for a cross memory post request. To ensure that all outstanding cross memory post requests for the ECB have completed, the user must invoke the SPOST macro. The ECB might or might not be posted, depending on existing conditions.

Synchronous cross memory post

You can specify LINKAGE=SYSTEM on a cross-memory POST. If you do, the system performs a synchronous POST, if possible. If it is not possible to do the POST synchronously, the system schedules an asynchronous POST. A synchronous POST means that the POST process completes before the system returns control to your program. A synchronous POST is possible only when the ECB is not an extended ECB (see "Writing POST exit routines" on page 47) or an EVENTS ECB and the target address space is swapped in and its LOCAL lock is available. A synchronous POST runs faster. However, in the event of an error condition, the system does not give the ERRET routine control and abnormally ends your program.

Return code 0 from POST with the LINKAGE=SYSTEM parameter indicates that the POST was synchronous. Return code 4 indicates that the POST was asynchronous. Return code 8 indicates that the POST was asynchronous, but that the ERRET routine can not be utilized.

The serialization method used to control modifications to an ECB depends on whether or not the ECB is waiting. If the ECB is not waiting (the high order bit of the ECB is off), it may be *quick posted* via the compare-and-swap instruction using the technique described in "Bypassing the POST routine." If the ECB is waiting (the high order bit of the ECB is on), the LOCAL lock serializes updates to the ECB.

Bypassing the POST routine

You can bypass the POST routine whenever the corresponding WAIT has not yet been issued if the wait bit is not on. In this case, a compare-and-swap (CS) instruction can be used to quick post the ECB. The compare operand should reflect the ECB content with the wait and post bits off, and the swap operand should have the post bit on and contain the desired post code. If the wait bit is on in the ECB, the CS will fail (giving a non-zero condition code), and the normal POST routine must be executed. If the wait bit is not on, the CS will, in effect, post the completion of the event. Note that holding the LOCAL lock does not eliminate the requirement to use the CS instruction. Figure 5 on page 47 demonstrates an example of how to *quick post* an ECB.

```
        L  RX,ECB          Get contents of ECB.
        N  RX,=X'3FFFFFFF' Turn off wait and post bits
        L  RY,=X'40000000' Post bit and post code
        CS RX,RY,ECB       Compare and swap to post ECB
        BZ POSTDONE        Branch if CS is successful
        LTR RX,RX          Wait bit on?
        BM DOPOST          If yes, then execute POST
        N  RX,=X'40000000' Is ECB posted?
        BNZ POSTDONE       If yes, do not execute POST
DOPOST  POST ECB
POSTDONE EQU  *
```

Figure 5. Bypassing the POST Routine

Waiting for event completion (EVENTS)

The EVENTS macro allows a user to wait for the completion of one of a series of events and be directly informed by the system which of the events have completed. Branch entry to this function, which is more efficient than SVC entry, is available to users executing in key 0, supervisor state, and holding only the LOCAL lock.

Branch entry is specified by coding BRANCH=YES on the EVENTS macro. If this parameter is used, the branch entry routine performs all normal WAIT processing and ECB initialization. You can specify BRANCH=YES in conjunction with either WAIT=YES, WAIT=NO, or ECB=.

- If you specify WAIT=YES, control will later be returned to the system, even though there might be ECBs posted to the EVENTS table. EVENTS frees the LOCAL lock. Before issuing the EVENTS macro with the WAIT=YES option, you must establish the return environment by setting the following:
  - RBOPSWA - Set with the PSW, which contains the address of the routine that is to receive control when the event completes
  - TCBGRS - Set to the general purpose registers (GPRs)

  EVENTS returns, in register 1, the address of the first processed event table entry or zero. (This service is not available to Type 1 SVCs or SRBs.)
- If you do not specify WAIT=YES, control returns to you. EVENTS does not free the LOCAL lock.

Writing POST exit routines

The POST exit function provides authorized routines with a service that allows them to receive control immediately upon each completion of an outstanding event (other POST functions can be used by unauthorized routines). Thus, the user can write a routine that receives control between the time the ECB is marked completed and the return by POST to the caller.

This function defines a special type of ECB known as an extended ECB. When initialized, these extended ECBs identify potential work requests rather than waiting tasks. A purpose of an extended ECB is to notify a process (for example, a subsystem) of an additional work request. Thus when an extended ECB is posted, a subroutine of the process receives control and updates a queue to identify the current work request.

When using the POST exit function, your routine must follow this sequence:

- Identify POST exit routines.

- Initialize extended ECBs and ECB extensions.
- Wait for work requests.
- Delete POST exit routines before terminating.

Identifying and deleting exit routines

Exit identification and deletion is performed through a function code that indicates whether the input exit address should be added to or deleted from the POST exit address queue for the current address space. A function code of 4 indicates an exit creation request, while 8 indicates an exit deletion request. Details of this interface are in “Branch Entry to the POST Service Routine”.

A 24-bit caller of the POST-exit-delete function can only delete an exit below 16 megabytes; a 31-bit caller must pass a valid 31-bit address and can delete an exit above or below 16 megabytes.

The process that establishes a POST exit is responsible for deleting that exit before its normal or abnormal termination.

Initializing extended ECBs and ECB extensions

The user must obtain and initialize the extended ECBs and ECB extensions. A system service is not available to perform these functions.

The ECB extension must be obtained and initialized before the initialization of the extended ECB. This sequence avoids the possibility of an initialized extended ECB being posted before the initialization of the ECB extension.

The ECB extension is two words long, begins on a word boundary, and can be from any subpool. However, the POST routine must be able to read from the ECB extension in the PSW key of the issuer of the POST macro. The ECB extension must also be accessible in the addressing mode of the POST's caller. More than one extended ECB can point to it. The mapping for the ECB extension is available via the EXT=YES parameter on the IHAECB mapping macro. The ECB extension has the format shown in Table 7:

Table 7. ECB Extension (ECBE)

ECB extension format		
VALUE (1 byte)	MODE (1 byte)	RESERVED (2 bytes)
	POST DATA (4 bytes)	

The fields in the ECBE are:

VALUE
is one byte containing a value of 1 to 255. A value of 1 indicates that the POST exit function is being requested.

MODE
The first bit of this byte indicates the addressing mode of the exit routine. If the byte contains X'80', the exit routine will receive control in 31-bit addressing mode. If the byte contains X'00', the exit routine will receive control in 24-bit addressing mode.

POST DATA
This field contains the address of the exit routine to be given control when the POST occurs.

The extended ECB must conform to current requirements for ECBs and be initialized as shown below. The extended ECB must be initialized only after it is eligible for posting. The extended ECB must be initialized using a compare-and-swap (CS) instruction. Holding the LOCAL lock does not eliminate the requirement to use the CS instruction to initialize the ECB because the ECB could be *quick posted* by a routine, using CS, that does not hold the LOCAL lock. It is unnecessary to hold the LOCAL lock to initialize an extended ECB. Compare and swap is necessary and sufficient to initialize the extended ECB. The meaning of the bits in the extended ECB follows:

Bits	**Meaning**
0	If one, indicates initialized ECB.
1-29	Bits 1-29 of the address of the associated ECB extension.
30-31	If ones, indicates an extended ECB.

If the compare and swap fails and if the ECB is pre-posted, the user should perform the appropriate POST exit functions in order to replace those ordinarily performed by the already concluded POST processing.

POST interface with exit routines

Before giving the exit control, POST checks to ensure that the user's exit routine address identified in the ECB extension denotes a valid POST exit routine. Even though POST thereby makes sure that a valid system exit receives control, the exit routine must ensure that an unauthorized routine has not counterfeited the extended ECB/ECB extension pair (by keeping the ECB/ECB extension pair in system key storage, for example).

The user's exit routine receives control from POST with the LOCAL lock in supervisor state, key zero. The routine must not release the LOCAL lock and should be able to process in both SRB and TCB mode. The register contents at entry to the user exit routine are:

Register	**Contents**
0	Address of the currently posted ECB
1	Address of related ECB extension
2-13	Unpredictable
14	Return address
15	Exit routine entry point address

Registers 11 and 14 must remain unchanged upon return to POST from the user's routine.

The user's exit routine must return control in supervisor state, PSW key zero and with the LOCAL lock still held.

For performance reasons, the user's routine should not cause page faults (that is, the routine's code and the data that it references should be in fixed or DREF storage when the routine receives control).

When you use an extended ECB on the POST macro and HASN (the home address space) doesn't equal PASN (the primary address space), use LINKAGE=SYSTEM with the ERRET parameter. The POST service then assumes that the extended ECB is in the primary address space and schedules an asynchronous cross-memory POST. The only way that such a POST can inform the requestor of an error condition is through an ERRET routine.

When you specify LINKAGE=SYSTEM, the POST macro service issues a return code. Return code 4 indicates that the system will perform the POST asynchronously. If you specified the ERRET parameter on the POST macro and an error occurs during asynchronous POST processing, MVS invokes the routine identified on the ERRET parameter. If your program is unauthorized and you specify ERRET, the POST is done asynchronously, but the error routine will not get control in the event of an error. Return code 8 indicates this condition.

Re-entry to POST from a POST exit

A POST exit routine cannot post another extended ECB unless it does so by specifying a cross memory post. Any attempt to activate another POST exit before the completion of the current exit causes a X'702' abend. If you must post another extended ECB from a POST exit routine, you should either have your routine issue a cross memory post or schedule your own SRB so that your routine enters POST by branching to it.

Branch entry to the POST service routine

Branch entry to the POST service routine provides, through LINKAGE=BRANCH, all the normal ECB and RB POST processing. To use the entry points, shown in Table 8 on page 51, you must write your own code. In general, the caller must hold the LOCAL lock and be in supervisor state, PSW key zero. Upon completion of the POST process, control returns to the caller in supervisor state, PSW key zero with the LOCAL lock.

Note: CML (cross memory local) lock means the local lock of an address space other than the home address space. LOCAL lock means the local lock of the home address space. When written in lower case, local lock means any local-level lock, either the local or a CML lock.

You can use branch entry to the POST service routine in cross memory mode for cross memory POST. If you hold the LOCAL lock of the home address space and if bit 0 of register 12 is 0, then the current address space must be the home address space and registers 0-9 and 14 are preserved. If you do not hold home's LOCAL lock or if bit 0 of register 12 is 1, then the current address space can be any address space and only registers 9 and 14 are preserved.

Note: If the high-order bit of register 12 is 0 and an error routine is invoked, the error routine is dispatched in the home address space. The error routine is also dispatched in the home address space when you use MEMREL=YES on the POST macro. However, if the high-order bit of register 12 is 1 (which is equivalent to coding MEMREL=NO on a POST macro), then the error routine is dispatched in ASID 1.

Table 8 shows the POST function and the branch entry points through which those functions can be performed. Table 9 shows the input parameters to POST. Table 10 shows the output parameters from POST.

Table 8. POST Function and Branch Entry Points

Functions		**Entry Points**		
	LINKAGE=BRANCH	CVT0PT02	CVT0PT03*	CVT0PT0E
Local ECB POST	X	X	X	
Cross memory POST	X**		X	
Post exit creation/deletion				X

* This entry point performs processing identical to entry point POST macro linkage=BRANCH. It is designed for use only by POST exit routines (that is, routines that receive control from POST as the result of having established that exit via entry point CVT0PT0E).

** The local lock does not need to be held for a cross memory POST at this entry point.

Table 9. POST Branch Entry Input

Registers	**CVT0PT02**	**CVT0PT03**	**CVT0PT0E**
0		ECB storage protect key(1)	Function Code
1			Exit Routine Address
10	Completion Code	Completion Code(2)	
11	ECB Address	ECB Address(3)	
12		Error Routine Address(4)	
13		ASCB Address(4)	
14	Return Address	Return Address	Return Address
15	Entry Point Address	Entry Point Address	Entry Point Address

Note:

1. If cross memory post, optionally contains the storage protection key of the ECB in bits 24-27.
2. If cross memory post and the storage protection key of the ECB is supplied in register 0, then the high order bit must be set to one.
3. If local POST, ensure high order bit of register is zero; if cross memory POST, set high-order bit of register to 1.
4. Only necessary when performing cross memory POST. If performing a cross memory POST and the high order bit in register 12 is on, only registers 9 and 14 are retained, and the error routine executes in the master scheduler's address space.

Table 10. POST Branch Entry Output

Entry Points/Options	**Registers Saved and Restored**
LINKAGE=BRANCH and ASCB not specified	0-9, 12, 13

Table 10. POST Branch Entry Output (continued)

Entry Points/Options	Registers Saved and Restored
LINKAGE=BRANCH and ASCB specified • Local lock held and MEMREL=YES • Local lock not held or MEMREL=NO	 0-9 9
CVT0PT02	0-9, 12-14
CVT0PT03	0-14
CVT0PT0E	2-14

Branch entry to the WAIT service routine

Branch entry to the WAIT service routine, which is not available to Type 1 SVCs or SRBs, provides all the normal ECB and RB WAIT processing. IBM does not recommend branch entry into the WAIT service routine because the caller must manually save the environment to be restored. A better choice might be to use the LINKAGE=SYSTEM parameter statement with the WAIT macro. See *z/OS MVS Programming: Assembler Services Reference IAR-XCT* for details on the SYSTEM=LINKAGE parameter with the WAIT macro.

While holding home's LOCAL lock and before branching to WAIT, the caller must establish the PSW and register return environment in its RB and TCB by setting the following:

- RBOPSWA - Set to the first and second half of the PSW, includes the key and mask bits as well as the address of the next instruction after the WAIT macro
- TCBGRS - Set to the general purpose registers (GPRs)

When WAIT is invoked, the caller should hold only the LOCAL lock. WAIT performs the following functions:

- Stores the ECB/ECBLIST address into the register 1 location of the TCB register save area (TCBGRS1). User data cannot be passed through this field or register.
- Releases home's LOCAL lock.
- Returns control to the system (control does not return to the caller even though all previously pending events have already occurred). The system ensures that all FRRs have been deleted.

The following registers contain parameters for branch entry to WAIT:

Register
Contents

0
The wait count in the low order byte. When the high order bit is one, it indicates long-wait (the LONG=YES specification).

1
The ECB pointer value. If only one ECB is being waited on, place that ECB address in register 1. If a list of ECBs is being waited on, place the complemented ECBLIST address in register 1.

15
The branch entry address to WAIT (obtained from the CVT field CVTVWAIT).

You can use branch entry to the WAIT service routine in cross memory mode if you hold the LOCAL lock of the home address space and if the current address space is the home address space.

Serializing RB processing

For request blocks (RBs) and service request blocks (SRBs), alternate serialization methods are available. Usually, these alternate methods provide better performance than do WAIT and POST processing. To serialize an RB routine, use the SUSPEND, CALLDISP, and RESUME macros. The SUSPEND macro identifies the RB that is to be suspended until an expected event occurs. The CALLDISP macro enables a routine that has suspended the current RB to call the system to have other work dispatched. After issuing the CALLDISP macro, the RB routine is suspended. When the event occurs that the suspended RB is waiting for, use the RESUME macro to tell the system that the suspended RB is again eligible to execute.

Serializing an SRB is described in "Serializing SRB processing" on page 208.

Suspending an RB until an event completes (SUSPEND)

The SUSPEND macro with the RB parameter provides an efficient way for an RB to wait for an event to complete. It is similar to the WAIT macro; the SUSPEND-RESUME sequence can be compared to the WAIT-POST sequence.

The SUSPEND macro does not have an immediate effect on the issuer as the WAIT macro does. Instead, the effect is delayed, depending on the type of suspension the macro user requests. If the previous RB is suspended, the effect takes place when the current RB returns to the system. If the current RB is suspended, the suspended state occurs when the RB passes control to the system.

RBs that issue the SUSPEND macro with the RB=CURRENT option should hold the suspended state time to a minimum. As soon as possible after SUSPEND completion, the RB that issues a SUSPEND RB=CURRENT should issue a CALLDISP macro with the BRANCH=YES option. Using the SUSPEND macro this way minimizes potential performance problems because the RB in this case must either be disabled or must hold the LOCAL lock or a CML lock. Minimizing suspension time also minimizes other potential problems the program might experience by limiting the time in which the RB is unable to cause any synchronous interrupts (such as SVCs and page faults) or provide interfaces to the WAIT, POST, or EVENTS macros.

RBs that issue SUSPEND RB=PREVIOUS, on the other hand, do not require the same synchronization because they are operating on behalf of another RB. The suspension of the previous RB does not require disabled execution or the holding of the LOCAL lock or a CML lock.

The following scenarios show typical SUSPEND macro sequences:

Scenario 1:

SUSPEND RB=PREVIOUS

1. Type 2 SVC routine receives control from a macro issuer.
2. The SVC routine suspends (through SUSPEND) the macro issuer's RB.
3. The process that will eventually issue the RESUME is started.
4. The SVC completes processing and exits.
5. Event completion occurs; process started in step 3 resumes (through RESUME) the issuer of the macro.
6. The macro issuer's task resumes (at return from the SVC routine).

Scenario 2:

SUSPEND RB=CURRENT

1. User acquires the LOCAL lock or a CML lock.
2. The macro suspends processing of the current RB.
3. The process that will eventually issue the RESUME is started.
4. Caller issues CALLDISP BRANCH=YES, which releases the LOCAL lock or CML lock.
5. Event completion occurs; process started in step 3 resumes issuer of the macro.
6. Normal processing resumes.

Considerations when suspending an RB

When using the SUSPEND macro with the RB parameter, there are several factors to consider:

General considerations:

- The SUSPEND macro can be issued in cross memory mode.
- Only a routine running in supervisor state with PSW key 0 can issue SUSPEND.
- The SUSPEND macro requires that the CVT mapping macro be included.
- Only task-related users can issue SUSPEND to suspend an RB, and then only for the current task.
- The SUSPEND function user must ensure that the SUSPEND and RESUME sequence takes place in proper order. The user must issue SUSPEND, then event completion must occur, and then the RESUME function must take place. One way to ensure proper sequencing is to issue SUSPEND before scheduling the asynchronous process on which the RB must wait.
- Do not intermix the SUSPEND and RESUME sequence with the WAIT and POST sequence on a single RB. Because the SUSPEND-RESUME sequence is a restricted-use function, it does only minimal validity checking. For example, if an RB were already waiting on 255 events and someone issued a SUSPEND against it, one POST or RESUME could make the RB dispatchable.
- An RB can have only one SUSPEND outstanding at a time. There can be no subsequent SUSPEND macros issued until a RESUME occurs for the outstanding SUSPEND macro.

RB=PREVIOUS considerations:

- When the issuer requests (explicitly or by default) the SUSPEND RB=PREVIOUS option, there must be a previous RB on the chain to prevent a task abend.
- SUSPEND RB=PREVIOUS is intended for use by Type 2, 3, and 4 SVCs to place the issuer of the SVC in a suspended state.

RB=CURRENT considerations:

- When using the SUSPEND RB=CURRENT option, the issuer must hold the CPU lock, LOCAL lock, or a CML lock. The issuer must remain in this state until the program initiates the stimulus for event completion in order not to lose control, which could result in never being redispatched. Because the issuer must also coordinate the SUSPEND and RESUME sequence, the event completion must not occur until after the SUSPEND RB=CURRENT macro takes effect. The caller that is in PSW key 0 supervisor state and EUT (enabled unlocked task) mode and that uses a local lock to serialize the SUSPEND and RESUME processing sequence can issue CALLDISP FRRSTK=SAVE. The CALLDISP routine releases the local lock, which serialized the SUSPEND/RESUME processing of the caller. Because an EUT FRR exists, the current FRR stack is saved.

- When a Type 1 or Type 6 SVC issues a SUSPEND RB=CURRENT, the top RB (the caller of the SVC) is suspended. Whenever the SVC exits (via EXIT PROLOGUE or T6EXIT), the caller is suspended until RESUME occurs. A TYPE 1 SVC must not issue the CALLDISP macro or release the LOCAL lock, and it must exit via its exit mechanism. A Type 6 SVC must not issue the CALLDISP macro or become enabled, and it also must exit via its exit mechanism.
- When a TYPE 2, 3, or 4 SVC issues a SUSPEND RB=CURRENT, the top RB (the SVC itself) is suspended. The SUSPEND routine returns control to the SVC. The SVC can continue to execute as long as it remains locally locked or disabled. Once the SVC releases the LOCAL lock or enables, an interrupt or a CALLDISP invocation suspends the SVC until it is resumed. While the SVC is enabled and before it is resumed, it cannot incur a page fault, issue an SVC, or branch enter any supervisor service that makes local work ready or places the caller in a wait state (for example, WAIT, POST, EVENTS, or STATUS).
- A program that is using a local lock or a CML lock to serialize SUSPEND RB=CURRENT must not release the lock before it issues CALLDISP. A program that uses the CPU lock to serialize SUSPEND RB=CURRENT must release the CPU lock immediately before it issues CALLDISP.

Using the CALLDISP macro

Routines that suspend the current RB can use the CALLDISP macro with the BRANCH=YES option to enter the suspended state. The BRANCH=YES option allows an issuer of the SUSPEND macro with its RB=CURRENT option to exit while leaving the current RB in the wait state. This option causes the system to save status and dispatches the next ready dispatchable unit.

Some considerations for using the BRANCH=YES option on the CALLDISP macro are:

- The issuer of CALLDISP must be executing in supervisor state with PSW key zero.
- The issuer must be in task mode.
- The issuer can be in cross memory mode.
- The BRANCH=YES option requires inclusion of the IHAPSA mapping macro.
- The FIXED=YES or FIXED=NO option can be specified with BRANCH=YES.
- When FRRSTK=SAVE is specified:
  - If the caller holds the LOCAL or CML lock, an enabled unlocked task (EUT) FRR must exist.
  - If any EUT FRRs exist, the current FRR stack is saved and the caller may hold either the LOCAL or CML lock. CALLDISP releases the lock.
- When FRRSTK=NOSAVE is specified:
  - The current FRR stack is purged.
  - The caller may hold either the LOCAL or CML lock. CALLDISP releases the lock.

Note:

1. A type 1 and type 6 SVC must not issue the CALLDISP macro.
2. The LOCAL or CML lock can be used to serialize the SUSPEND processing and establish the RESUME processing.

Resuming execution of a suspended RB

The RESUME macro, which is supported in cross memory mode, provides an efficient means for indicating the completion of an event. The RESUME macro specifies the TCB and RB that were previously suspended by the SUSPEND macro. The specified TCB and RB must be addressable in the currently addressable address space. Only routines executing in supervisor state and PSW key zero can issue the RESUME macro.

The RESUME macro and the service routine it calls must serialize the use of the task that is being resumed. This serialization might require the local lock of the task's address space, called the target address space. Because disabled or locked callers of RESUME are not allowed to obtain a local lock, the RESUME macro has the MODE and ASYNC options to handle these types of situations.

Note: The ASYNC parameter for the RESUME macro is spelled differently from similar parameters on other macros.

The MODE option specifies whether or not the RESUME operation must complete (MODE=UNCOND) or not (MODE=COND).

The ASYNC option specifies whether or not RESUME can schedule an SRB to perform the resume if necessary. These RESUME options can be combined in four ways:

- MODE=UNCOND and ASYNC=N
  - RESUME attempts to obtain the necessary task serialization to complete the function synchronously. If it can obtain serialization, RESUME completes its function and returns to its caller. If it cannot obtain serialization, RESUME requests the local lock of the target address space to serialize the operation. The caller of RESUME must be able to obtain the target address space's local lock or already hold it when RESUME is issued. This means that, with one exception, the caller of RESUME must either be running enabled and unlocked or must hold the target address space's local lock when the RESUME is issued. The exception is the disabled caller that resumes the TCB under which it is running, that is, the currently executing TCB. This situation could occur if, for example, a routine became disabled, executed a SUSPEND RB=CURRENT macro, and then determined that there was more work to be done. The disabled, unlocked routine could issue a RESUME macro for the TCB and RB to counteract the SUSPEND.

    If the local lock is required but not available, the caller will be suspended waiting for the local lock. Control returns to the caller after the RESUME has occurred.
  - Disabled interrupt exits cannot issue the RESUME macro with the MODE=UNCOND and ASYNC=N options.
  - The RETURN=N option on the RESUME macro is allowed only with this combination of options. The RETURN=N option cannot be used with the ASCB parameter. To use RETURN=N, the caller must be running in SRB mode, must be in primary ASC mode where primary=home, and must not hold any locks. If these three conditions are met, the system transfers control to the task that was just resumed. If these three conditions are not met, that caller is abended with an X'070' abend code.
- MODE=UNCOND and ASYNC=Y

  - RESUME attempts to obtain the necessary task serialization and complete the function synchronously. If RESUME cannot obtain serialization, RESUME does not obtain the local lock. RESUME unconditionally schedules an SRB to complete the RESUME asynchronously.
  - The caller can be enabled or disabled.
- MODE=COND and ASYNC=N
  - RESUME attempts to obtain the necessary task serialization to complete the function synchronously. If serialization is available, the task is resumed and control returns to the caller. If serialization is not available, RESUME returns to the caller without completing the RESUME operation.
  - The caller can either be enabled or disabled and can hold any combination of locks. RESUME does not attempt to obtain any locks. The caller must be prepared to handle the situation when the RESUME operation cannot be performed because the necessary serialization is not available.
- MODE=COND and ASYNC=Y
  - RESUME attempts to obtain the necessary serialization to complete the function synchronously. If serialization is available, the task is resumed and control returns to the caller. If serialization is not available, RESUME might schedule an SRB to complete the RESUME asynchronously.
  - The caller can be either enabled or disabled, and can hold any combination of locks. RESUME does not attempt to obtain any locks. The caller must be prepared to handle the situation when the RESUME operation cannot be performed because the necessary serialization is not available and the RESUME could not be processed asynchronously.

RESUME provides return codes in register 15 to indicate the result of the RESUME attempt. See the RESUME macro in *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU* for details on the return codes.

The RESUME macro requires that you include the IHAPSA mapping macro. If the ASCB option is not specified, then the MODE=UNCOND and ASYNC=Y combination requires that you include the CVT mapping macro.

Synchronizing unit of work (tasks or SRBs)

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

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

Table 11. Pause, Release, and Transfer callable services

Callable service	64-bit version callable service	Description
IEAVAPE, IEAVAPE2	IEA4APE, IEA4APE2	Obtains a pause element token (PET), which uniquely identifies a pause element (PE).
IEAVDPE, IEAVDPE2	IEA4DPE, IEA4DPE2	Frees a pause element (PE) that is no longer needed.
IEAVPSE, IEAVPSE2	IEA4PSE, IEA4PSE2	Pauses the current task or SRB.
IEAVRLS, IEAVRLS2	IEA4RLS, IEA4RLS2	Releases a paused task or SRB.

Table 11. Pause, Release, and Transfer callable services (continued)

Callable service	64-bit version callable service	Description
IEAVRPI, IEAVRPI2	IEA4RPI, IEA4RPI2	Retrieves information about a pause element (PE).
IEAVTPE	IEA4TPE	Tests a pause element (PE) and determines its state.
IEAVXFR, IEAVXFR2	IEA4XFR, IEA4XFR2	Releases a paused task and, when possible, gives it immediate control, while optionally pausing the task under which the Transfer request is made.

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

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

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

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

Pause elements and pause element tokens

A pause element (PE) is a system-managed object used to pause and release a work unit, which can be either a task or SRB. Like an ECB, a PE is used by the system to control whether or not a work unit is dispatchable. You can use a PE, like an ECB, to prerelease a work unit before it is paused. There are, however, significant differences between an ECB and a PE. Table 12 compares the two.

Table 12. Pause Element (PE) and Event Control Block (ECB)

Pause Element (PE)	Event Control Block (ECB)
Managed by the system.	Managed by application programs.
Identified by a pause element token (PET).	Identified by a simple address.
Can change the dispatchability of the task or SRB in any address space.	Unless specifically allocated in common storage, can change the dispatchability of only a task in the same address space as the ECB. Even when allocated in common storage, can change the dispatchability of only a task, not an SRB.
Cannot be reused once invalidated by an asynchronous ABEND.	Can be reused after a task is removed from the wait state by an asynchronous ABEND (or other RTM dispatchability management), although reuse requires very complicated programming.

Table 12. Pause Element (PE) and Event Control Block (ECB) (continued)

Pause Element (PE)	Event Control Block (ECB)
To prevent errors related to reuse, a PET can be used only once to identify a PE. Once a task or SRB is paused and released, an updated PET is needed to identify the PE. The system returns the updated PET to the caller through the Pause or Transfer service.	There is no control on the reuse of an ECB; the same ECB address can be used over and over.
PEs are allocated and deallocated through calls to system services.	ECBs are allocated by application programs.
The total number of PEs available to all unauthorized programs in a system is limited to 130560.	The system imposes no limits on the number of ECBs.
PEs are addressable only through PETs. The user of a PE does not know its actual address and cannot modify it except through the use of system services.	ECBs can be modified at will by application programs.

A PE allocated by an unauthorized caller can be used to pause and release any work unit in the caller's home address space. An unauthorized caller cannot pause or release using a PE allocated with auth_level=IEA_AUTHORIZED or pause_element_auth_level=IEA_AUTHORIZED.

When a PE is allocated with auth_level=IEA_AUTHORIZED (callable services IEAVAPE and IEA4APE) or pause_element_auth_level=IEA_AUTHORIZED (callable services IEAVAPE2 and IEA4APE2), the PE can be used to pause and release any task or SRB in the system. The same PE can be used, for example, to pause a task in address space 10. After being released, the same PE can be used to pause an SRB in, say, address space 23. There is, however, a small performance penalty imposed when a PE is used to pause a task or SRB in one space and then reused to pause a task or SRB in another space. This cost is accrued upon each space transition.

The following services (IEAVAPE2, IEAVDPE2, IEAVPSE2, IEAVRLS2, IEAVRPI2, IEAVXFR2, IEA4APE2, IEA4DPE2, IEA4PSE2, IEA4RLS2, IEA4RPI2, and IEA4XFR2) have a 'LINKAGE=' keyword that specifies whether the caller is authorized (LINKAGE=BRANCH requires the caller to be in key 0 supervisor state). The authorization of the PET is *not* specified, except when it is allocated via the IEAVAPE2 (or IEA4APE2) service. Instead, the authorization of the PET is implied. Authorized users can use any PET, but unauthorized users only can use a PET that was allocated for unauthorized use.

When an authorized program is passed a PET from an unauthorized program, the authorized program must ensure that the PET is valid to be used by the unauthorized program. The authorized program can validate this by invoking the IEAVRPI2 (or IEA4RPI2) service and specifying a special value for the linkage parameter. When an authorized program needs to validate that a PET can be used by an unauthorized program, add IEA_UNTRUSTED_PET to the value specified for LINKAGE and invoke the IEAVRPI2 service. The data returned by the IEAVRPI2 service identifies whether the unauthorized caller is able to use the PET:

- Return_code must be 0.
- Pause_element_auth_level must be IEA_PET_UNAUTHORIZED.

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

Pause elements are not supported by checkpoint/restart. However, support has been added with APAR OA19821 to permit the allocation of pause elements that can allow a successful checkpoint/restart. This is possible for applications that can tolerate the fact that the pause element will not be restored upon a restart after a checkpoint. An application can tell if this support is available by checking if CVTPAUS4 has been set in the CVT. A new type is introduced for the auth_level parameter of the Allocate Pause Element service. The new level is created by adding the value of the new type (IEA_CHECKPOINTOK) to the existing value (at one time, only IEA_AUTHORIZED or IEA_UNAUTHORIZED can be specified). When pause elements are allocated, a checkpoint is accepted only if all encountered pause elements have been allocated indicating the IEA_CHECKPOINTOK type.

Using the services

| There are 26 callable services available for task synchronization:

- Allocate_Pause_Element – IEAVAPE, IEAVAPE2, IEA4APE, or IEA4APE2
- Deallocate_Pause_Element – IEAVDPE, IEAVDPE2, IEA4DPE, or IEA4DPE2
- Pause – IEAVPSE, IEAVPSE2, IEA4PSE, or IEA4PSE2
- Release – IEAVRLS, IEAVRLS2, IEA4RLS, or IEA4RLS2
- Retrieve_Pause_Element_Information - IEAVRPI, IEAVRPI2, IEA4RPI, or IEA4RPI2
- Test_Pause_Element - IEAVTPE or IEA4TPE
- Transfer – IEAVXFR, IEAVXFR2, IEA4XFR, or IEA4XFR2

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

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

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

When you are finished with the PE, call the Deallocate_Pause_Element service to return the PE to the system. If a work unit is asynchronously ABENDed while it is paused, the system itself invalidates the PE, and it cannot be reused for pause requests. Thus, return an invalidated PE to the system as soon as possible by a call to Deallocate_Pause_Element.

Though the PE remains allocated until you deallocate it, you can use a PET only once, to pause and release a work unit. When you specify a PET on a successful call to the Pause service or to pause a work unit through a successful call to the Transfer service, the system invalidates the input PET and returns an updated PET to identify the PE. Use the updated PET to reuse the PE or to deallocate the PE.

Figure 6 on page 61 shows, in pseudocode, the sequence of calls to allocate a PE, pause the current work unit, release the work unit, and deallocate the PE.

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

Figure 6. Pause and Release Example

The Pause, Release, and Transfer services also provide a release code field that is particularly useful when a dispatchable unit might be released before it is paused. The program that calls the Release service can set a release code.

Release code is returned to the invoker of Pause and can be used to communicate the reason for the release.

Figure 7 on page 62 shows, in pseudocode, the sequence of calls needed to allocate a PE, prerelease a work unit, and deallocate the PE

```
/* Common variables              */  |
                                     |
 Dcl PET char(16);                   |
                                     |
    Workunit #1                      |     Workunit #2
                                     |
 /* Workunit #1 variables        */  | /* Workunit #2 variables       */
 Dcl Auth1 fixed(32);                | Dcl Auth2 fixed(32);
 Dcl RC1  fixed(32);                 | Dcl RC2   fixed(32);
 Dcl Updated_PET char(16);           | Dcl RelCode binary(24);
 Dcl RetRelCode binary(24);          |
                                     |
 Auth1 = IEA_AUTHORIZED;             |
                                     |
 /* Allocate a Pause Element      */ |
 Call IEAVAPE (RC1,Auth1,PET);       |
            .                        | Auth2 = IEA_AUTHORIZED;
            .                        | RelCode='123';
            .                        |
                                     | /* Release Workunit #1         */
                                     | Call IEAVRLS (RC2,Auth2,PET,
                                     |               RelCode);
 /* Pause Workunit #1             */ |            .
 Call IEAVPSE (RC1,Auth1,PET,        |            .
            Updated_PET,RetRelCode); |            .
                                     |
/*check release code and continue */ |
            .                        |
            .                        |
            .                        |
 /* Deallocate the pause element  */ |
 Call IEAVDPE (RC1,Auth1,            |
            Updated_PET);            |
```

Figure 7. Release and Pause Example

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

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

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

Figure 8 on page 63 shows an example of using the Transfer service to release a dispatchable unit without pausing the current work unit.

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

1. The current pause element token (CurrentDuPet in Figure 8 on page 63) identifies the allocated pause element that Transfer is to use to pause the current dispatchable unit (the caller of the Transfer service). When you do not need to pause the current dispatchable unit, you set this token to binary zeros, as shown in Figure 8 on page 63.

2. The updated pause element token (UPET2 in Figure 8), which the system returns when you specify a current pause element token. You need this updated token to reuse the pause element on a subsequent Pause or Transfer or to deallocate the pause element. If you set the current token to binary zeros, as done in Figure 8, the contents of the updated pause element token are not meaningful.
3. The target token (TargetDuPET in Figure 8) identifies the allocated pause element that Transfer is to use to release a dispatchable unit. In Figure 8, it contains the PET that identifies the PE used to pause Workunit #1.

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

```
/* Common variables             */    |
                                      |
 Dcl PET char(16);                    |
                                      |
        Workunit #1                   |  Workunit #2
                                      |
/* Workunit #1 variables        */    | /* Workunit #2 variables        */
 Dcl Auth1  fixed(32);                | Dcl Auth2  fixed(32);
 Dcl RC1    fixed(32);                | Dcl RC2   fixed(32);
 Dcl UPET1 char(16);                  | Dcl CurrentDuRelCode char(3);
 Dcl RetRelCode binary(24);           | Dcl CurrentDuPET binary(24);
          .                           | Dcl UPET2 char(16);
          .                           | Dcl TargetDuPET  char(16);
          .                           | Dcl TargetDuRelCode binary(24);
Auth1 = IEA_AUTHORIZED;               |
/* Allocate a Pause Element      */   | Auth2 = IEA_AUTHORIZED;
 Call IEAVAPE (RC1,Auth1,PET);        |           .
                                      |           .
/* Pause Workunit #1            */    |           .
 Call IEAVPSE (RC1,Auth1,PET,UPET1,   | TargetDuRelCode = '123';
              RetRelCode);            | /* no pause-set token to zeros  */
                                      | CurrentDuPet ='';
                                      | TargetDuPET =  PET
/*processing pauses until transfer*/  |
                                      | /* Transfer to Workunit #1      */
                                      |  Call IEAVXFR (RC2,Auth2,
                                      |               CurrentDuPET,UPET2,
                                      |               CurrentDuRelCode,
                                      |               TargetDuPET,
                                      |               TargetDuRelCode);
/*processing continues          */    |           .
                                      |           .
 /* Deallocate the Pause Element */   |           .
 Call IEAVDPE (RC1,Auth1,UPET1);      |
```

Figure 8. Transfer without Pause Example

PE ownership and cleanup

A PE has an address space relationship used to define when PEs are automatically deallocated by the system. When a PE is originally allocated, it is owned by the home address space at the time of the call. When an authorized caller pauses a task or SRB, the ownership of the PE moves to the home address space of the task or SRB that is being paused. This change in ownership occurs even if the PE has been prereleased. When the highest level task in an address space terminates, the system automatically deallocates all of the PEs the address space owned.

Because PEs allocated by unauthorized callers cannot move between address spaces, the ownership defined at allocation time remains unchanged for the life of the PE.

Global resource serialization latch manager

The global resource serialization latch manager is a service that authorized programs can use to serialize resources within an address space or, using cross memory capability, within a single MVS system. Programs can call the latch manager services while running in task or service request block (SRB) mode.

The latch manager functions are callable services for all systems application architecture (SAA) high level languages. The callable services allow applications to efficiently serialize shared or exclusive access to resources. For example, an application can use several latches to serialize resources within an address space, instead of using the address space local lock to serialize those resources. Other features, such as high speed, make the latch manager a flexible alternative to the ENQ/DEQ macros and the locking function. For a comparison of these serialization services, see “Choosing a serialization service” on page 25.

This information contains the following topics:
- An overview of the function of the latch manager
- Planning to use the latch manager
- A guide to the latch manager callable services.

Overview

Note: In this topic, the term *program* refers to a task or service request block (SRB) routine.

The main advantage to using the latch manager is that you can reduce contention for resources by dividing them into smaller segments and assigning individual latches to those segments. Your application can assign latches to resources in whatever way it desires. The more you can divide resources and assign individual latches to them, the more likely it is that you will reduce contention for those resources and increase the overall performance of your application.

For example, if a requestor needs to update a data structure that contains several sub-structures, and those sub-structures are used by several programs running concurrently in the same address space, the requestor could obtain the address space local lock to serialize access to the entire data structure. Because there is only one address space local lock, the system suspends other requestors until the lock is released. On the other hand, if your application were to assign a latch to each sub-structure (there is no system-imposed limit on the number of latches you can assign to resources), requestors could serialize access to individual sub-structures within the data structure at the same time.

“How to use the callable services” on page 65 explains how to use the latch manager services to serialize resources that your application uses. Before reading the information, understand the following terms:

Latch A structure that the latch manager uses to track and handle requests to obtain, release, and purge latches. The application associates each latch with a resource, then requests access to the resource by calling the Latch_Obtain or the 64-bit Latch_Obtain service with the appropriate latch number.

Latch identity
An attribute that consists of thresholds, a printable description of the latch's usage, and other attributes as defined by the latch identity entry in the appropriate language related macro - ISGLMC, ISGLMASM, or ISGLMPLI.

Latch set
A set of latches that an application can use to serialize its required resources. The application calls the Latch_Create or the 64-bit Latch_Create service to create a latch set.

Latch set creator
An application that calls the Latch_Create or the 64-bit Latch_Create service to create a set of latches. Programs that run as part of the application can use the latches to serialize required resources.

Latch set token
An 8-byte area that identifies a latch set. The latch manager returns the latch set token to the caller of the Latch_Create or the 64-bit Latch_Create service.

Latch token
An 8-byte area that identifies an individual latch request. The latch manager returns the latch token to the caller of the Latch_Obtain or the 64-bit Latch_Obtain service.

Requestor
A program that calls a latch manager callable service to:

- Request ownership of a latch, or
- Release an owned latch, or a pending request to obtain ownership of a latch.

How to use the callable services

To use the latch manager, your application should call the latch manager services in the following sequence:

1. **Create a latch set.** Your application begins by calling the Latch_Create service to create a set of latches in the current primary address space. Each latch can then be used to serialize resources. Your application can assign latches to resources in whatever way it desires.
2. **Provide latch identities for latches.** You might choose to use the latch identity service (ISGLID/ISGLID64) to create latch identities and other attributes for your latches. Latches whose attributes are not changed by the usage of the obtainer can be created right after the latch set is created. Latches whose attributes are changed by usage of obtainer can still be initialized to a "not in use state" right after the latch set is created. The attributes of latches changed by obtainers should be updated or replaced by ISGLID/ISGLID64 before and after usage with appropriate values.
3. **Request ownership of latches.** Programs that run as part of your application can call the Latch_Obtain service to request exclusive or shared ownership of latches associated with required resources. If there is no contention for a latch at the time of the Latch_Obtain request, the latch manager can grant ownership

of the latch immediately. The caller of Latch_Obtain specifies a parameter that tells the latch manager what to do when a latch is not immediately available, which is one of the following:

- Queue the request and suspend the requestor until the latch manager eventually grants ownership of the latch to the requestor
- Do not queue the request; return control to the requestor
- Queue the request, return control to the requestor, then post an event control block (ECB) when the latch manager grants ownership of the latch to the requestor.

When a requestor obtains ownership of a latch, the requestor can read or modify the associated resource, depending on the type of ownership that was requested (shared or exclusive).

4. **Release ownership of owned latches or pending latch requests.** After a requestor finishes using the resource, it calls the Latch_Release service to release the latch so other requestors can use the resource. Requestors can also call Latch_Release to release a pending request to obtain ownership of a latch that has been queued by the latch manager.
5. **If an error occurs, purge all owned latches and pending requests for a particular requestor.** Latch_Purge, which is similar to Latch_Release, is a convenient way to purge all granted and pending requests for a specific requestor within a specific latch set. The latch manager provides the Latch_Purge service for use by an application's termination resource manager.

“Planning to use the latch manager callable services” describes how to set up your application to use the latch manager callable services.

Planning to use the latch manager callable services

Before using the latch manager to serialize resources for your application, you should perform the following tasks:

Task	Reference
1. Determine how the application is to serialize resources using the latch manager services: **a.** Identify the resources that your application needs to serialize. **b.** Assign latches to those resources in such a way that contention for those resources is minimized. Determine the latch attributes to use for each latch set and when the attribute needs to be changed. Determine if any latch deadlock detections should be specified for each latch set and what level the detection is.	“Creating a latch set (ISGLCRT and ISGLCR64 services)” on page 71“Specifying a latch's identity or usage” on page 72
2. Include or copy the appropriate latch manager interface definition file (IDF) into programs that invoke the latch manager services.	“Including a latch manager interface definition file (IDF)” on page 67

Task		Reference
3.	Linkedit a linkage assist routine into each module that calls the latch manager services (optional).	"Loading the linkage assist routines"
4.	Provide recovery for the latch manager.	"Providing recovery for the latch manager" on page 68

There are some considerations to be aware of when using enclaves for tasks that serialize resources using the latch manager callable services. For details, see "Using ENQ/DEQ or Latch Manager Services With Enclaves" in *z/OS MVS Programming: Workload Management Services*.

Once you have completed the tasks, you can begin to use the latch manager services described in "Guide to the latch manager callable services" on page 70.

Including a latch manager interface definition file (IDF)

The latch manager provides files, called IDFs, that define variables and values for the parameters used with latch manager services. IDFs are available for different languages. You can include or copy an IDF from a central library into programs that invoke the latch manager services.

The following IDFs are available on MVS:

Table 13. Latch Manager IDFs

Language	Contained in	Member
Assembler H	SYS1.MACLIB	ISGLMASM
C/370™	SYS1.SAMPLIB	ISGLMC
PL/I	SYS1.SAMPLIB	ISGLMPLI

IBM does not provide IDFs for COBOL, Pascal, or FORTRAN.

Loading the linkage assist routines

Your application can linkedit the latch manager linkage assist routines into each module that calls the latch manager services. Alternatively, your application can load the linkage assist routines to obtain their entry points, and then branch directly to them.

The following linkage assist routines are contained in SYS1.CSSLIB:

- ISGLCRT
- ISGLCR64
- ISGLID
- ISGLID64
- ISGLOBT
- ISGLOB64
- ISGLREL
- ISGLRE64
- ISGLPRG
- ISGLPR64

Providing recovery for the latch manager

When you use the latch manager to serialize resources, your application must provide recovery to handle errors that can occur when the application code or the latch manager is in control. For performance reasons, and because SRB routines might not want to release owned latches when errors occur, the system does not release owned latch requests during task or address space termination. The system does, however, release pending latch requests when the requestor's home address space terminates. The recovery you provide can be either an ESTAE-type or a functional recovery routine (FRR). The following example shows pseudocode for an application that calls the latch manager services and its associated FRR.

Guideline: The procedure that follows shows the exact method for ensuring recovery for the Latch_Obtain and Latch_Release functions.

For information about how to provide a resource manager recovery routine, see Chapter 18, "Providing recovery," on page 327.

Steps for providing recovery for the latch manager

About this task

Before you begin: You **must** use this procedure to ensure recovery for the Latch_Obtain and Latch_Release functions. Not following this exact procedure can risk the loss of serialization during recovery.

In the "Program mainline example" on page 69, which follows this procedure, the mainline program performs the following tasks:

Procedure

1. Issue the SETFRR macro to establish the recovery routine.
2. Initialize:
   a. Initialize the latch_token to zeroes as a guide to knowing if the latch manager has potentially obtained the latch. The latch manager will place a latch token value into the area specified on the latch_token parameter by the caller of the Latch_Obtain service. However, unless the service returns, the recovery routine does not know if the latch was definitely obtained.

      Note: A nonzero latch token value does not necessarily indicate that the caller has definitely obtained the latch, it indicates that there is a potential for holding the latch.

   b. Initialize the updating_resource flag to null. This is the bit that indicates whether or not you definitely hold the latch and whether you started to potentially perform incomplete updates to the serialized data.
   c. Initialize the init_complete flag, which indicates whether the values set by the application are valid and can be trusted by the recovery routine. The init_complete flag must reside in storage that was initialized to zeroes before the program is called or before the recovery routine is established.
3. Set the obtain_option field to ISGLOBT_SYNC (value of 1). If the latch manager cannot immediately grant ownership of the latch, the latch manager suspends the requestor, and returns control to the requestor when ownership of the latch is eventually granted.

   Note: The latch manager sets the latch token value prior to waiting for the latch to become available, so the token can be nonzero in recovery cases when the latch is not actually held.

4. Call the Latch_Obtain service to obtain a latch. At this point, if recovery is entered due to an error in Latch_Obtain services, it is still uncertain if the latch is held.
5. Turn the updating_resource flag “on” to indicate that the latch is now definitely held and that the program is about to update the resource associated with the obtained latch.
6. Update the resource.
7. Turn the updating_resource flag “off” to indicate that the resource update is complete and the Latch_Release service is about to be called. Once this flag is turned off, there is uncertainty about whether the latch is definitely held.
8. Set the release_option field to ISGLREL_UNCOND (value of 0). If the original caller of Latch_Obtain no longer owns the latch, this value tells the system to abnormally end the caller of Latch_Release. The abend is desired because the resource was potentially updated without the proper serialization and consequently corrupted.
9. Call the Latch_Release service to release the latch. The latch token should have the value that was returned from the obtain services. Note that the Latch_Release service does not reset the token to zero.

FRR tasks

In the “FRR example” on page 70, which follows this procedure, the FRR performs the following tasks:

1. Tests the init_complete flag to determine if the updating_resource and latch_token variables are initialized.
2. Tests the updating_resource flag. If the flag is “on”, the latch is definitely held and the FRR can repair the resource associated with the currently obtained latch. This is how the FRR knows if the resource is serialized.
3. Tests the latch_token to see if it is a non-zero value. If so, the FRR assumes that the program might have obtained the latch; therefore, the FRR should call Latch_Release with the ISGLREL_COND (value of 1) option to ensure the latch is no longer held. ISGLREL_COND (value of 1) indicates that the latch manager should return control to the requestor with a return code instead of issuing an abend if the latch is not held.

Program mainline example: Use this example to help set up recovery for the latch manager.

```
SETFRR                               /* Establish recovery routine    */
latch_token=Zeroes;                  /* Clear latch token             */
updating_resource=OFF;               /* Clear resource update flag    */
init_complete=ON;                    /* Set initialization flag       */
CALL ISGLOBT(latch_set_token,        /* Obtain the latch associated   */
             latch_number,           /* with resource to be updated   */
             requestor_ID,
             ISGLOBT_SYNC,           /* Indicates the resource should */
             ISGLOBT_EXCLUSIVE,      /* wait for the resource to be   */
             ECB_address,            /* available
             latch_token,            /* see note 1 below              */
             work_area,              /* see note 2 below              */
             return_code);
updating_resource=ON;                /* Indicate that the program is  */
        .                            /* about to update the resource  */
 < Update resource >                 /* Code for resource update      */
        .
updating_resource=OFF;               /* Resource update complete      */
CALL ISGLREL(latch_set_token,        /* Release the latch             */
             latch_token,
             ISGLREL_UNCOND,        /* You want ISGLREL to abend     */
             work_area,             /* if the latch was not owned    */
             return_code);
```

Note:

1. An ABEND can occur prior to the service returning, consequently the latch token might be set by the service even when the latch is not obtained.
2. Zero is the only possible return code for the ISGLOBT_SYNCH option, so you know that you definitely hold the latch.

FRR example

Example: Use this example to help set up recovery for the latch manager.

```
IF init_complete=ON THEN             /* Is initialization complete?   */
DO
 IF updating_resource=ON THEN        /* Resource possibly updated?    */
   DO                                /* Note: If updating_resource    */
        .                            /* bit is on, the latch is held  */
        .
  < Repair Resource >                /* Code for restoring resource   */
        .
   END;
 IF latch_token ¬= 0  THEN           /* Is latch owned? (the latch    */
   DO                                /* could still be held, or the   */
        .                            /* request could still be queued)*/
     ISGLREL(latch_set_token,        /* Release the latch, if owned   */
             latch_token,            /* Use conditional release option*/
             ISGLREL_COND,           /* since latch may not be owned  */
             work_area,
             return_code);
        .
   END;
END;
< Continue recovery >                /* Continue as appropriate       */
```

Guide to the latch manager callable services

This information introduces the latch manager services, describes the information you must provide when calling each service, and offers suggestions about when to use each service.

The latch manager provides the following callable services for all SAA high level languages:

Table 14. Latch Manager Services

Service	Call Name	Description
Latch_Create	ISGLCRT	Create a set of latches.
Latch_Create	ISGLCR64	Create a set latches in 64-bit mode.
Latch_Identify	ISGLID	Identify a latch.
Latch_Identify	ISGLID64	Identify a latch in 64-bit mode.
Latch_Obtain	ISGLOBT	Obtain exclusive or shared control of a latch.
Latch_Obtain	ISGLOB64	Obtain exclusive or shared control of a latch in 64-bit mode.
Latch_Release	ISGLREL	Release ownership of an owned latch or a pending request to obtain a latch.

Table 14. Latch Manager Services (continued)

Service	Call Name	Description
Latch_Release	ISGLRE64	Release ownership of an owned latch or a pending request to obtain a latch in 64-bit mode.
Latch_Purge	ISGLPRG	Purge all granted and pending requests for a particular requestor within a specific latch set.
Latch_Purge	ISGLPR64	Purge all granted and pending requests for a particular requestor within a specific latch set while in 64-bit mode.

The syntax for calling the latch manager callable services in assembler language is:

```
CALL routine_name,(parm1,parm2,...return_code)
```

Note: This information references equate symbols and associated numeric values that are defined in the latch manager interface definition files (IDFs). (IDFs define variables and values for the parameters used with latch manager services.) Each equate symbol is followed by its numeric equivalent; for example, "ISGLREL_COND (value of 1)" indicates the equate symbol ISGLREL_COND and its associated value, 1. You may specify either the equate symbol or its associated value when coding the latch manager callable services. For more information about latch manager IDFs, see "Including a latch manager interface definition file (IDF)" on page 67.

Creating a latch set (ISGLCRT and ISGLCR64 services)

Applications can call the Latch_Create service to create a latch set during initialization or whenever latches are required. The system keeps each latch set in storage with a protection key that matches the PSW key of the caller of Latch_Create.

The caller of Latch_Create must specify:

- The number of latches to be contained in the latch set (see "Specifying the number of latches in a latch set" on page 72).
- A name that uniquely identifies the latch set within the current primary address space (see "Identifying latch sets" on page 72).

If you want the latch obtain services to detect some simple latch deadlock situations, you can specify the ISGLCRT_DEADLOCKDET1 or ISGLCRT_DEADLOCKDET2 option on the ISGLCRT service. The following deadlock situations can be detected:

- The work unit requests exclusive ownership of a latch that the work unit already owns exclusively.
- The work unit requests shared ownership of a latch that the work unit already owns exclusively.
- The work unit requests exclusive ownership of a latch for which the work unit already holds a shared ownership.

See *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for instructions about how to use the two deadlock detection options.

Once you create a latch set, the latch set remains in place for the life of the job or address space. You cannot delete a latch set.

Specifying the number of latches in a latch set

The number of latches that you include in each latch set depends on the number of latches that your application requires and the number of latch sets you decide to create. For example, if you estimate that your application will require 100 latches, you could create a single latch set that contains all the required latches, two latch sets that contain 50 latches each, and so on. You should create your latch sets and assign latches to resources so as to minimize contention for those resources.

If you find that your application requires more latches after you have created one or more latch sets, you must create an additional latch set to add the latches; you cannot add latches to an existing latch set.

If you need to associate latches with an extremely large number of resources, you can create a large latch set and use a hashing routine to associate the resources with latches. You may want to use this method if you have resources that can dynamically change, such as data sets, programs, or address spaces.

The amount of storage consumed by a latch set in the latch set creator's address space varies depending on the z/OS release level, the highest number of concurrent outstanding latch obtains, and the latch set create options. The following values are approximate for z/OS V1R9 and for planning purposes only:

- Each latch set is 128 bytes plus 32 bytes per latch and 110 bytes per outstanding latch obtain request. Additional working storage is used to enhance performance when the system uses the default latch set create option (that is, when the ISGLCRT_LOWSTGUSAGE create option is *not* specified). The amount of additional working storage used by each latch set varies with the number of configured processors.
- If the ISGLCRT_LOWSTGUSAGE create option is used instead of the default option, less working storage is consumed at the expense of performance. Consider using the ISGLCRT_LOWSTGUSAGE option and fewer latches when the latch set creator's address space is constrained by private storage.

Identifying latch sets

When calling Latch_Create, your application must specify a name for the latch set to be created. The latch set name uniquely identifies the latch set within the current primary address space. If you specify a latch set name that already exists in the current primary address space, the Latch_Create service sets the return code to indicate that the latch set name already exists, and provides the latch set token for the existing latch set.

IBM recommends that you use a standard naming convention for the latch set name. To avoid using the names IBM uses for its products, do not begin the latch set name with the character string "SYS." Also, ensure that the latch set name will be readable when it is listed in output from the DISPLAY GRS command and interactive problem control system (IPCS). Use of national characters ('@', '$', '#') is discouraged because those characters display differently on different national code pages. For more information about displaying latch information with DISPLAY GRS, see *z/OS MVS System Commands*. For more information about using IPCS to display latch contention information and statistics, see *z/OS MVS IPCS Commands*.

Specifying a latch's identity or usage

The resource manager of the application can call the ISGLID and ISGLID64 service to identify how an individual latch is being used at any point in time. The ISGLID and ISGLID64 service provide a latch set creator with the ability to attach a latch identity array (in ISGLMASM or ISGLMC) to the latch set. By using a LIDArray, individual latch usage attributes including a description and thresholds can be identified. The latch attributes are defined by the latch owner. Depending on the dynamic nature of each latch, the latches usage attributes might never change and are assigned once or are initiated to not in use, changed after obtaining the latch, and reset to not in use before releasing the latch. The effect on performance should always be evaluated when using dynamic latch attributes. Ensure that LIDArray is in the primary space.

After the LIDArray has been attached to the latch set, it cannot be deleted. However the Latch Identity Pointer Array can be replaced by calling the service again and specifying a new array. To change the Latch identity for a particular latch, allocate a new latch identity block, fill it out, and update the corresponding array entry. Then the program is free to delete the previous latch identity block.

The Latch identity provides attributes that are related to the current usage of a given latch. The attribute includes threshold values and a printable description of the usage of the latch. The description is displayed on the D GRS,ANALYZE,LATCH command to provide the meaning of a specific latch usage. The thresholds are used by the system to determine when contention or hold times are abnormal. Non-zero threshold values should be carefully planned as users of the D GRS,ANALYZE,LATCH command may not see latches reported as in contention until the specified latch threshold values have been exceeded.

The identity can also be used to find more detailed component information about the latches usage. Any displayable text can be used. See *z/OS MVS Diagnosis: Reference* for how other components have provide diagnosis information for the latches.

See the following example for how to specify the thresholds and printable string.

```
LIDHOLDTHRESHOLD DS F
```

The time in seconds that is not normal for this latch to be held. With this field, a value of X'00000000' is deemed to mean "take the default". This allows users to not have to specify the default value of X'FFFFFFFF' in every entry. The default value indicates that no matter how long this latch is held, it should not be treated as an exception. An example usage of this field would be for an application to issue a message if it were to find a latch held for a time period longer than that specified here, even if no contention exists.

```
LIDCONTTHRESHOLD DS F
```

The time in seconds that is not normal for this latch to be in contention. For example, when D GRS,ANALYZE,LATCH,BLOCKER and D GRS,ANALYZE,LATCH,WAITER process this element, it will be ignored unless the current longest waiter’s wait time exceeds this threshold. A value greater than X'1D55600' (one year) will be treated by these commands as an indication to never display this latch.

```
LIDPRINTABLESTRING@ DS AD
```

Address of printable (EBCDIC) string

```
LIDPRINTABLESTRINGLENGTH DS H
```

Length of printable (EBCDIC) string

For more information about how to use the ISGLID service, see the ISGLID and the ISGLID64 topics in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Obtaining a latch (ISGLOBT and ISGLOB64 services)

Once your application creates one or more latch sets, requestors can call the Latch_Obtain service to obtain latches for exclusive or shared access to resources. Requestors must call Latch_Obtain from the same primary address space from which the Latch_Create service was called to create the associated latch set.

When a requestor calls Latch_Obtain, the latch manager does one of the following:

- If the latch is available, grants ownership to the requestor immediately
- If the latch is not immediately available, performs an action specified by the requestor (see “Specifying an obtain option” for descriptions of the various actions).

In cases where the latch manager finds contention for a requested latch, the Latch_Obtain service performs more slowly than in cases where latch manager can grant ownership of the latch to the requestor immediately.

Specifying the number of a requested latch

When your application creates a latch set that contains the latch you wish to obtain, the latch manager assigns a number to each latch in the latch set. Callers of the Latch_Obtain service must specify this number when requesting ownership of a specific latch.

The system numbers the latches starting from zero. The requestor specifies the requested latch on the latch_number parameter on the call to Latch_Obtain. The highest numbered latch is the total number of latches in the latch set minus one.

Specifying an obtain option

When a requestor calls the Latch_Obtain service to request ownership of a latch, the latch manager cannot grant control of the latch immediately if contention exists because either:

- *Exclusive* ownership of the latch was requested (Latch_Obtain was called with an access option of ISGLOBT_EXCLUSIVE, value of 0), and an exclusive or shared request (either owned or pending) already exists for the latch
- *Shared* ownership of the latch was requested, (Latch_Obtain was called with an access option of ISGLOBT_SHARED, value of 1), and an exclusive request (either owned or pending) already exists for the latch.

When either situation occurs, the value specified on the obtain_option parameter indicates how the latch manager is to handle the request, which is one of the following:

- **ISGLOBT_SYNC (value of 0):** Queue the request and suspend the requestor; when the latch manager eventually grants ownership of the latch to the requestor, return control to the requestor
- **ISGLOBT_COND (value of 1):** Do not queue the request; return control to the requestor
- **ISGLOBT_ASYNC_ECB (value of 2):** Queue the request and return control to the requestor; post an ECB when the latch manager eventually grants ownership of the latch to the requestor.

Summary of results of calls to Latch_Obtain

The following table summarizes the results of calls to the Latch_Obtain service:

Table 15. Summary of Possible Results of Calls to Latch_Obtain

Situation	Obtain Option	Access Option	Resulting Actions for Caller of Latch_Obtain	Outcome of Request to Obtain Latch
Latch is not owned	Any	Any	Return control to caller	Ownership granted immediately; if obtain option ISGLOBT_ASYNC_ECB (value of 2) is specified, ECB is not posted
Only shared requestors own latch	Any	ISGLOBT_SHARED (value of 1)	Return control to caller	Ownership granted immediately; if obtain option ISGLOBT_ASYNC_ECB (value of 2) is specified, ECB is not posted
Only shared requestors own latch	ISGLOBT_SYNC (value of 0)	ISGLOBT_EXCLUSIVE (value of 0)	Suspend caller	Request queued, ownership granted when latch available
Only shared requestors own latch	ISGLOBT_COND (value of 1)	ISGLOBT_EXCLUSIVE (value of 0)	Return control to caller	Request not queued
Only shared requestors own latch	ISGLOBT_ASYNC_ECB (value of 2)	ISGLOBT_EXCLUSIVE (value of 0)	Return control to caller	Request queued, ECB posted when ownership granted
An exclusive requestor owns or is waiting for the latch	ISGLOBT_SYNC (value of 0)	Any	Suspend caller	Request queued, ownership granted when latch available
An exclusive requestor owns or is waiting for the latch	ISGLOBT_COND (value of 1)	Any	Return control to caller	Request not queued
An exclusive requestor owns or is waiting for the latch	ISGLOBT_ASYNC_ECB (value of 2)	Any	Return control to caller	Request queued, ECB posted when ownership granted

Releasing a latch (ISGLREL and ISGRE64 services)

Requestors can call the Latch_Release service to release an owned latch or a pending request to obtain a latch. It is important to release latches when they are no longer needed so other requestors can obtain those latches.

When processing a Latch_Release request, the latch manager runs more slowly if contention for the latch exists at the time of the call to Latch_Release. To allow the latch manager to release latches most efficiently, you should design your application to minimize contention for resources.

It is possible that a requestor might end (either normally or abnormally) without releasing an owned latch or a pending request to obtain a latch. To make such a latch available, code in your application's recovery can call the Latch_Release service. Your application's recovery can also call the Latch_Purge service to purge all granted and pending requests for a particular requestor within a specific latch set. See “Providing recovery for the latch manager” on page 68 for additional information.

Specifying a release option

When calling Latch_Release, the requestor specifies the latch token that was returned on the previous Latch_Obtain call. The latch token identifies the specific request to be released. On the call to Latch_Release, the requestor also specifies a release option, which tells the latch manager to do one of the following if the latch manager did not grant ownership of the latch to the caller of Latch_Obtain:

- **ISGLREL_UNCOND (value of 0):** Abnormally end the caller of Latch_Release.
- **ISGLREL_COND (value of 1):** Return control to the caller with a return code; the remaining actions depend on the obtain_option specified on the original call to the Latch_Obtain service.

“Summary of results of calls to Latch_Release” describes the results of calls to Latch_Release.

Summary of results of calls to Latch_Release

The following table summarizes the outcomes from calls to the Latch_Release service:

Situation	Release Option	Resulting Actions for Caller of Latch_Release	Outcome of Request to be Released
Latch is owned	Any	Return control to caller	Request released, new owner possible
Requestor suspended	ISGLREL_UNCOND (value of 0)	Abend caller	Request remains queued
Requestor suspended	ISGLREL_COND (value of 1)	Return control to caller	Request remains queued
Requestor is waiting on an ECB	ISGLREL_UNCOND (value of 0)	Abend caller	Request remains queued
Requestor is waiting on an ECB	ISGLREL_COND (value of 1)	Return control to caller	Request released
Request not found	ISGLREL_UNCOND (value of 0)	Abend caller	No action on the latch
Request not found	ISGLREL_COND (value of 1)	Return control to caller	No action on the latch

Purging one or more latches (ISGLPRG and ISGLPR64 services)

Your application's resource manager can call the Latch_Purge service to purge all granted and pending requests for a particular requestor within a specific latch set. Latch_Purge is particularly useful when a task or address space is terminating, and you want to release latches that were obtained by a terminating task or programs that are running in a terminating address space. For information about how to call Latch_Purge, see “Providing recovery for the latch manager” on page 68.

Purging one or more latches in a group of latch sets for a group of requestors (ISGLPBA and ISGLPB64 services)

Your application's resource manager can call the Latch_Purge_by_Address_Space service to purge all granted and pending requests for a group of requestors within a group of latch sets. Latch_Purge_by_Address_Space is particularly useful when a task or address space is termi- nating, and you want to release latches that were obtained by a terminating task or programs that are running in a terminating address space.

To effectively use this service the latch_set_names and your requestor_IDs should be defined such that they have a common portion and a unique portion. Groups of latch sets can then be formed by masking off the unique portion of the latch_set_name and groups of latch requests in a latch set can then be formed by masking off the unique portion of the requestor_ID. This technique is very useful when keeping track of the unique portion of the requestor_ID might be difficult or expensive. A single purge request to can then handle multiple latch sets and multiple requests in a latch set. Recovery routines should call Latch_Purge_by_Address_Space when one or more errors prevent requestors from releasing latches.

Purging groups of latch sets

To better understand how latch sets names and requestor IDs might be grouped, look at the following example. Suppose that you have created three latch sets in an address space with the following names:

```
LATCHSET1 DC CL48'EXAMPLE.ONE_LATCH_SET_1'
LATCHSET2 DC CL48'EXAMPLE.ONE_LATCH_SET_2'
LATCHSET3 DC CL48'EXAMPLE.ONE_LATCH_SET_3'
```

There are multiple tasks doing spaced-switched program calls (PCs) into this address space which will be obtaining and releasing latches from these latch sets. For a requestor_ID, the following is used:

```
MVC  REQ_IDH,PSAAOLD-PSA     /* use home ASCB address and   */
MVC  REQ_IDL,PSATOLD-PSA     /* TCB address as requestor ID */
```

The first word of the requestor_ID contains the ASCB address of the home address space and the second word is the address of the TCB. For each address space, that has tasks using the latch sets, an end-of-memory resource manager is set up. If you are also using SRBs that would obtain and release latches you could use a similar scheme for the requestor IDs. For instance, the first word could be the ASCB and the second word could be the address of some control block in common (so that the address would be unique and different from the TCB address used by the non-SRB requestors). This is especially useful when the unique portion of the requestor_ID may not be readily determinable by the latch purge requestor.

When one of these address spaces terminates it may still hold one or more of the latches. When the resource manager gets control (it will be running in the master address space), it PCs into our address space to issue the ISGLPBA service. The following example sets up the parameters for this service:

```
     USING RMPL,R4
     MVC   REQ_IDH,RMPLASCB      /* use the terminating ASCB addr */
     MVC   REQ_IDL,ZEROS         /* rest of req_id should be zero */
```

The requestor_ID mask would then be:

```
    REQID_MK DC    4X'FF'        /* match terminating ASCB */
                   XL4'0'        /* mask off TCB address - */
    *                            /* purge latches for      */
    *                            /* for all tasks          */
```

The latch set name would be the common portion followed by zeros for the unique part:

```
    LSNME    DC    CL22'EXAMPLE.ONE_LATCH_SET_'  /* common part */
             DC    XL26'0'                       /* unique part */
```

And the mask would be:

```
    LSNME_MK DC    22X'FF'             /* common part */
             DC    XL26'0'             /* unique part  */
```

And the call to the service will be:

```
    CALL ISGLPBA,(LS_TOKEN,REQ_ID,REQID_MK,           X
                 LSNME,LSNME_MK,RETCD),MF=(E,PBYA_DPL)
```

Note that LS_TOKEN should be set to zero for the latch_set_name and latch_set_name_mask to be used.

```
    LS_TOKEN DC    XL8'0'             /* use latch set name and */
    *                                 /* latch set name mask    */
```

For each latch set in the primary address space, the latch_set_name will be logically ANDed with the latch_set_name_mask:

```
  latch_set_name     'EXAMPLE.ONE_LATCH_SET_1_..................'

                      CECDDDC7DDC6DCECC6ECE6F4444444444444444444444444
                      57147355655D31338D253D10000000000000000000000000
```

AND

```
 latch_set_name_mask  FFFFFFFFFFFFFFFFFFFFFF00000000000000000000000000
                      FFFFFFFFFFFFFFFFFFFFFF00000000000000000000000000
                      ________________________________________________

 result:             'EXAMPLE.ONE_LATCH_SET_..........................'
                      CECDDDC7DDC6DCECC6ECE600000000000000000000000000
                      57147355655D31338D253D00000000000000000000000000
```

When the result is compared to the latch_set_name operand, LSNME there is a match and the service will try to purge latches in this latch set.

Similarly, for the requestor ID, suppose a requestor in the terminating address space has obtained a latch with the following requestor ID:

```
ASCB   TCB

00FA7880008FD8E0
```

The service looks at all the requests in this latch set by ANDing the requestor ID with the requestor ID mask and comparing it to the requestor ID operand:

```
requestor_ID:          00FA7880008FD8E0
requestor_ID_mask:     FFFFFFFF00000000
                       ________________
result:                00FA788000000000
```

When the result is compared, to the requestor ID operand, there is a match and the latch request will be purged.

Chapter 5. Reporting system characteristics

This information describes the following ways to report system characteristics:
- Using ISGQUERY and GQSCAN to obtain resource usage reports
- Using SRM to obtain subsystem measurement reports
- Using IEAMRMF3 to obtain information about the dispatchability of address spaces
- Using an installation-written component trace to obtain information about component events.

ISGQUERY is the IBM recommended replacement for the GQSCAN service.

Collecting information about resources and their requestors (ISGQUERY and GQSCAN macros)

Global resource serialization enables an installation to share symbolically named resources. Programs issue the ISGENQ, ENQ and RESERVE macros to request access to resources; global resource serialization adds information about each requestor to the appropriate resource queue. The only way you can extract information from the resource queues is by using the ISGQUERY or GQSCAN macro. See *z/OS MVS Planning: Global Resource Serialization* for information about how global resource serialization functions.

Using ISGQUERY and GQSCAN, you can inquire about a particular scope of resources (such as STEP, SYSTEM, or SYSTEMS), a specific resource by name, a specific system's resources, a specific address space's resources, or resources requested through the RESERVE macro. The system collects the information you request from the resource queues and consolidates that information before returning it. The authorized calls return a “snapshot” of the outstanding global resource serialization requests. The system serializes the resource queue so it does not change while the system gathers the information. (Authorized calls are those that specify either LOCAL or GLOBAL on the SCOPE parameter.) The information returned by unauthorized calls, which do not perform serialization, might be inconsistent because of changes in the resource queue that can occur while the system collects the information. (Unauthorized calls are those that specify either STEP, SYSTEM, SYSTEMS, or ALL on the SCOPE parameter.)

Using ISGQUERY the system returns information you request about the status of each resource identified to global resource serialization, which includes information about the tasks that have requested the resource. ISGQUERY fully supports 64-bit callers and as of z/OS V1R6 is the IBM recommended replacement for the GQSCAN service. Use the ISGQUERY service to inquire about:
- A particular scope of resources (such as STEP, SYSTEM, or SYSTEMS).
- A specific resource by name.
- A specific system or systems resource.
- A specific address space resource.
- Resources that are requested through the RESERVE macro.
- Requests that specified Userdata.

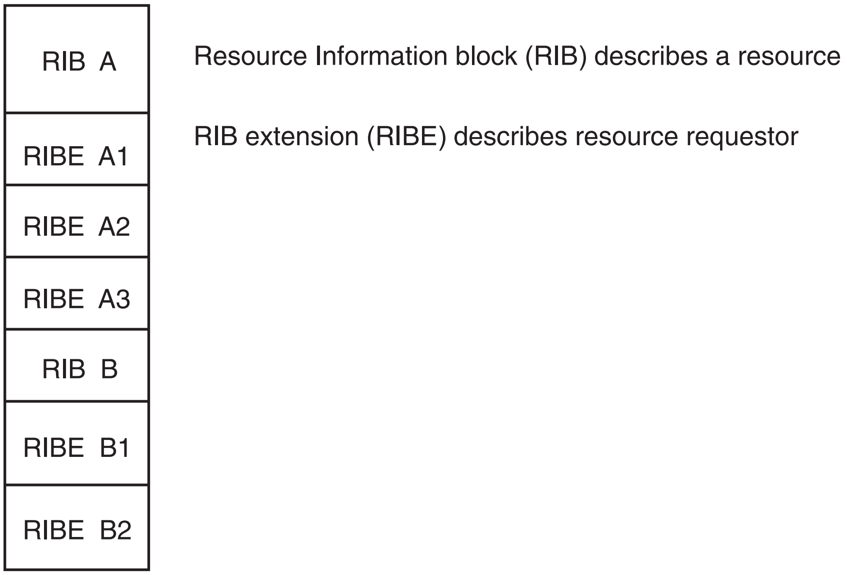
The system collects the information you request from the resource queues and consolidates that information before returning it. The ISGQUERY service returns the following types of global resource serialization information:

- **REQINFO=RNLSEARCH:** To determine if a given resource name is in the current Resource Name Lists (RNL).
- **REQINFO=QSCAN:** To obtain information on resources and requesters of outstanding ENQ requests.
- **REQINFO=ENQSTATS:** To obtain information related to ENQ counts.
- **REQINFO=LATCHECA:** To obtain enhanced contention analysis data for latch waiters that might indicate contention issues.
- **REQINFO=USAGESTATS:** To obtain information for address space level contention information related to ENQs (all scopes) and latches (all latch sets).

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

How GQSCAN returns resource information

GQSCAN returns the information in the form of resource information blocks (RIB) and resource information block extensions (RIBE), as shown below. For the description of RIB and the RIBE, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).


RIB A
Resource Information block (RIB) describes a resource
RIBE A1
RIB extension (RIBE) describes resource requestor
RIBE A2
RIBE A3
RIB B
RIBE B1
RIBE B2

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

- RIBTRIBE contains the total number of RIBEs associated with this RIB
- RIBNRIBE contains the total number of RIBEs associated with this RIB that GQSCAN could fit into the area specified on the AREA parameter.

- RIBEDEVN contains a 4-digit EBCDIC device number for RESERVEs issued on the system. For RESERVEs issued on other systems, RIBEDEVN contains zero.

If the value in RIBNRIBE is less than the value in RIBTRIBE, you may need to specify a larger area with the AREA parameter.

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

How area size determines the information GQSCAN returns

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

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

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

How scope and token values determine the information GQSCAN returns

Table 16 and Table 17 on page 83 summarize the possible values and the information returned for a GQSCAN request.

Table 16. GQSCAN Results with a Scope of STEP, SYSTEM, SYSTEMS, or ALL

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

Table 16. GQSCAN Results with a Scope of STEP, SYSTEM, SYSTEMS, or ALL (continued)

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

The example in Figure 9 shows the area contents for three requests. For each request, the caller specified the TOKEN parameter and one of the following for the scope value: STEP, SYSTEM, SYSTEMS, or ALL. Assume that the resource queue contains information about four resources: A, which has three requestors; B, which has six; C, which has two; and D, which has one.

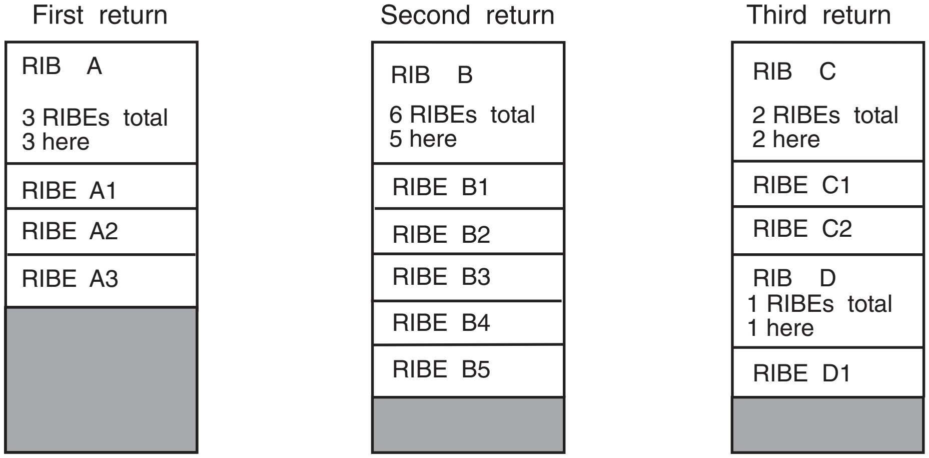


Figure 9. Work Area Contents for GQSCAN with a Scope of STEP, SYSTEM, SYSTEMS, or ALL

Note that, because the specified area is not large enough, the caller cannot receive all of the RIBEs associated with resource B, even though the caller coded the TOKEN parameter. To receive all of those RIBEs, the caller has to specify a larger area and reissue the GQSCAN request. Authorized callers (those who can specify SCOPE=LOCAL or SCOPE=GLOBAL) do not need to alter the size of the area they provide, as long as they specify the TOKEN parameter; they can receive additional RIBEs on subsequent calls to GQSCAN.

Table 17. GQSCAN Results with a Scope of LOCAL or GLOBAL

GQSCAN Invocation	TOKEN Parameter Coded?	Information Returned
Initial call	No	At least the first RIB that represents the first requestor on the resource queue, and as many of that RIB's associated RIBEs as will fit. Any RIBEs that do not fit are not returned to the caller. If all of the RIBEs fit, GQSCAN returns the next RIB on the resource queue, as long as the remaining area is large enough to hold that RIB and at least one of its RIBEs.
Initial call	Yes; value is zero	At least the first RIB that represents the first requestor on the resource queue, and as many of that RIB's associated RIBEs as will fit. Any RIBEs that do not fit are not returned unless you issue GQSCAN again, with the TOKEN parameter equal to the value that GQSCAN returns on this call. If all of the RIBEs fit, GQSCAN returns the next RIB on the resource queue, as long as the remaining area is large enough to hold that RIB and at least one of its RIBEs.
Subsequent call	No	At least the first RIB that represents the first requestor on the resource queue, and as many of that RIB's associated RIBEs as will fit. Any RIBEs that do not fit are not returned to the caller. If all of the RIBEs fit, GQSCAN returns the next RIB on the resource queue, as long as the remaining area is large enough to hold that RIB and at least one of its RIBEs.
Subsequent call	Yes; value is the token value returned by GQSCAN on the preceding call	If GQSCAN was unable to return all of the RIBEs for the last RIB returned on the preceding call, the information returned is that last RIB and its remaining RIBEs, plus any additional RIBs and their associated RIBEs as will fit. Otherwise, GQSCAN returns at least the next RIB on the resource queue and as many of that RIB's associated RIBEs as will fit.

The example in Figure 10 on page 84 shows the area contents for three requests. For each request, the caller specified the TOKEN parameter and one of the following for the scope value: LOCAL or GLOBAL.

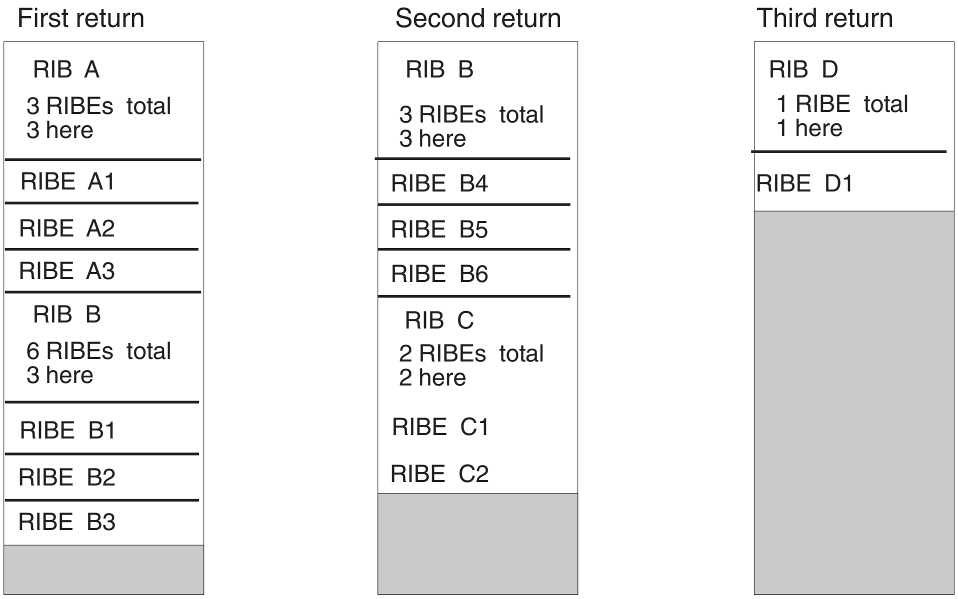


Figure 10. Work Area Contents for GQSCAN with a Scope of LOCAL or GLOBAL

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

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

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

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

How global resource serialization determines the scope of an ENQ or RESERVE request

Global resource serialization might change the scope of an ISGENQ, ENQ or RESERVE request that was issued with a SCOPE of SYSTEM or SYSTEMS (unless the request specified RNL=NO).

If an ENQ is issued with a scope of SYSTEM, and matches an entry in the SYSTEM inclusion RNL but does not match an entry in the SYSTEMS exclusion RNL, then global resource serialization changes the scope of the request from SYSTEM to SYSTEMS (and processes the resource as a global resource). A GQSCAN with SCOPE=SYSTEM or SCOPE=LOCAL will not find a match on such a request.

If an ENQ or RESERVE is issued with a scope of SYSTEMS, and either it matches an entry in the SYSTEMS exclusion RNL, or the system is running with GRSRNL=EXCLUDE, then GRS changes the scope of the request from SYSTEMS to SYSTEM (and processes the resource as a local resource). A GQSCAN with SCOPE=SYSTEMS or SCOPE=GLOBAL will not find a match on such a request.

When issuing a GQSCAN for a resource whose scope might have been changed, you might find it desirable to specify SCOPE=ALL in order to match requests whether or not they have been changed.

Using the SRM reporting interface to measure subsystem activity

The reporting interface allows an interactive subsystem to pass transaction performance data to the system resources manager (SRM). The data collected by the SRM can be reported through the RMF™ workload activity report or the transaction activity report. The Resource Measurement Facility (RMF) program product must be installed to obtain these reports. The data is reported according to the subsystem identifiers (subsystem name, transaction name, transaction class, accounting information, and user ID).

The reporting interface is necessary because, except for TSO/E, the SRM does not normally recognize the individual transactions of an interactive subsystem. For example, the SRM considers a subsystem that consists of an address space created by a START command to be a single long transaction, and the RMF workload activity report indicates the total service for the address space but does not indicate the average transaction response time. However, when a subsystem uses the interface and the subsystem is specified in the installation control specification, the RMF workload activity report provides the average transaction response time.

The reporting interface consists of a SYSEVENT macro using a mnemonic operand of TRAXRPT, TRAXFRPT, or TRAXERPT, which the subsystem must issue at the completion of each transaction. Issuing the macro allows the subsystem to pass the transaction start time or elapsed time and, optionally, its resource utilization. The SRM does not use data collected through the reporting interface to dynamically adjust resource distribution to subsystems. However, the installation can review the RMF reports to determine which, if any, SRM parameters need to be changed.

Obtaining dispatchability data about address spaces (IEAMRMF3 macro)

The IEAMRMF3 macro provides the following information about address spaces:

- How many address spaces are currently running on a processor
- Whether a specific address space is running on a processor
- How many address spaces are dispatchable, but not currently running on a processor.

For information about dispatchability of enclaves, use the IWMRQRY macro. For information about how to use IWMRQRY, see *z/OS MVS Programming: Workload Management Services*.

Before invoking IEAMRMF3, your program must obtain storage in which the system returns the dispatchability data. The returned information is an array of elements, each of which represents an address space; the array is mapped by the IHADSD mapping macro. For each potential address space in the system, your program can check two bits in the element to determine whether the address space:

- Is running on a processor
- Is dispatchable, but not actually running on a processor.

An address space can have both bits set to one. For example, both bits are set to one for a multitasking address space where at least one task is running on a processor and at least one task is dispatchable, but not running on a processor.

An address space can have both bits set to zero. Both bits are set to zero for an address space that does not exist, or an address space that is swapped out, or an address space with no dispatchable work.

You can search through the array in one of two ways:

- Use an ASID as an index into a specific element in the array.
- Use the value in the DSDINDXF field (in the header) as the index to the first element that represents an address space that has dispatchable work. From that element, use the value in the DSDINDXN field to find the next element, and so forth through the array. In this way, you look at only those elements that represent address spaces that have dispatchable work.

See *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for information about coding the IEAMRMF3 macro and an example of searching through an array for dispatchability data. For the structure of the IHADSD mapping macro, see the DSD data area in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Chapter 6. Tracing applications using component trace

The component trace service (also referred to in the remainder of this information as simply "component trace") provides common tracing services for components, subsystems, or applications. Component trace allows you to capture information about application-specific events for debugging problems in your application, or for system analysis.

To use component trace, you create trace entries at various points in your application to capture the data you require. You decide the location of the trace points, what the trace entries will contain, and which trace entries, if any, to create at any given time. You store the trace entries in trace buffers. The entries are externalized in a dump or, by using the component trace external writer, in a trace data set on a direct access storage device (DASD) or tape.

Benefits of using component trace: While you might choose to write your own trace services to capture the events, the advantages of using component trace instead are that you will:

- Have a consistent interface for all application traces.
- Avoid writing duplicate code.
- Eliminate the need to IPL or to use the SPZAP utility to turn traces on or off; the component trace service allows you to turn traces on or off dynamically.
- Be able to merge your trace data with other application traces, IBM-supplied component traces, and GTF user traces.
- Be able to merge different traces from the same system or from multiple systems in a sysplex.
- Be able to use IPCS to:
  - Print or view the trace
  - Combine the trace data set with another trace data set
  - Extract specific entries from a trace data set
  - Merge formatted trace output with other formatted trace output

This information contains the following:

- **"Planning an application trace" on page 88** tells what you need to do to establish a component trace for your application, including how to request it and how to obtain the trace records.
- **"Coding macros for application traces" on page 103** contains information about coding component trace macros that enable your application to use component trace:
  - Use the CTRACE macro to do one or more of the following:
    - Define your application to component trace
    - Delete your application from component trace
    - Specify a parmlib member containing trace options
    - Define multiple traces for your application
    - Indicate that your application supports the component trace external writer.

    Note:

1. For the full range of services provided by the CTRACE macro, see *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.
2. All references to the external writer in this information refer to the component trace external writer.

– Use the ITTCTE mapping macro when creating trace entries.

– When you are using the external writer:
  - Use the CTRACECS macro to manage trace buffers.
  - Use the CTRACEWR macro to write your trace buffers to DASD or tape.

- **"Coding a start/stop exit routine" on page 112** tells how to write a component trace start/stop exit routine. You must have a start/stop routine to use component trace. The start/stop routine starts, stops, or modifies tracing for the application by controlling the creation of trace entries by the application.
- **"Coding a display trace exit routine" on page 121** describes how to write a component trace display exit routine. The display routine provides status information about the component trace for the display trace operator command. This routine is optional.
- **"Creating trace entries" on page 125** describes how to create trace entries.

Planning an application trace

This information describes the design considerations needed to implement component trace. Decisions are needed for the following tasks to provide application tracing:

You Decide:	Where to Look:
When to trace and whether to provide minimum options (MINOPS) that cannot be turned off.	See "When to trace" on page 92.
Where the trace points will be and what the trace entries will contain.	See "Where and what to trace" on page 93.
How to create and control trace buffers that contain trace entries.	See "Creating trace buffers" on page 94.
Whether to provide multiple traces, which are called ***sublevel traces***, and how many traces your installation will need.	See "Using multiple traces" on page 96.
Whether to set up user-defined trace options. Component trace allows you to categorize your trace entries and create only those you select at a given time.	See "Setting up user-defined options" on page 96.
Whether a display exit routine is required.	See "Coding a display trace exit routine" on page 121.
How to provide default tracing options	See "Starting, stopping and changing the trace" on page 97.
How you will externalize your data. You can: • Request a dump • Combine the trace with other trace output • Use the component trace external writer (called the external writer in the remainder of this text) to write data directly to a data set.	See "Externalizing trace data in a dump" on page 99, or "Externalizing trace data through the external writer" on page 100.

You Decide:	Where to Look:
How you will format your data once it is externalized. • Component trace supports formatting through IPCS. • If you do not use IPCS, you will have to format the data yourself.	See *z/OS MVS IPCS Customization* and *z/OS MVS IPCS Commands* for information on using IPCS.

Understanding the Sequence of Trace Activities: The following is a simplified description of tracing using component trace.

- The programmer plans and sets up the user-defined control structures.
- The application defines itself to component trace.
- The operator turns the trace on, either because of a problem or because of a message from the application.
- The system passes control to the start/stop exit routine to start tracing.
- The application starts tracing by creating trace entries and placing them in trace buffers.
- The application or operator (or both) dumps the trace buffers.
- The operator turns off the trace.
- The system passes control to the start/stop exit routine to stop tracing, and optionally dump the trace buffers.
- The application deletes itself from component trace.
- The system programmer formats the trace buffers through IPCS.

Using component trace involves a number of trace-related activities. The following tables summarize the trace-related activities and how to accomplish each activity.

Trace activities

Trace-related activity:	How to accomplish the activity:
Define the application to component trace	CTRACE macro with DEFINE parameter (referred to as CTRACE DEFINE for the remainder of this text)
Start tracing	A user-written start/stop exit routine and one of the following: • TRACE CT and accompanying REPLY operator commands • Parmlib member on a TRACE CT operator command • PRESET in a parmlib member on a TRACE CT operator command • Parmlib member on CTRACE DEFINE
Create trace table entries	ITTCTE mapping macro
Obtain status information	DISPLAY TRACE operator command. A user-written display trace exit routine is optional.

Trace-related activity:	**How to accomplish the activity:**
Change trace options	A user-written start/stop exit routine and one of the following: • TRACE CT and accompanying REPLY operator commands • Parmlib member on a TRACE CT operator command **Note:** Specify MOD=YES on CTRACE DEFINE if you want to allow options to be changed while the trace is running.
Externalize trace data through a dump	• SVC dump requested by an SDUMPX macro (you can also use the SDUMP macro) • Stand-alone dump requested by the operator
Externalize trace data through the external writer	See "Externalizing trace data through the external writer" on page 100.
Stop tracing	A user-written start/stop exit routine and one of the following: • TRACE CT operator command • Parmlib member on a TRACE CT operator command • PRESET in a parmlib member on a TRACE CT operator command • Parmlib member on CTRACE DEFINE
Format the trace entries	• IPCS subcommands – CTRACE – COPYTRC – MERGE • CTRACE format table (ITTFMTB macro) • CTRACE buffer find exit routine (used when you place your trace buffers in a dump) • CTRACE filter/analysis (CTRF) exit routine (optional) • Trace entry format model
Delete the application from component trace	CTRACE macro with the DELETE parameter

Executable macros for component tracing

Macro	**Purpose or Related Activity**
CTRACE	• Define an application to component trace. • Delete an application from component trace.
SDUMPX	Dump the storage containing the trace buffers.
ITTFMTB	Create a format table.
CTRACECS	Manage trace buffers and determine when they are full.
CTRACEWR	Pass full trace buffers to external writer.

Note: See *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* through *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for more information.

Operator commands for component tracing

Operator Command	Purpose or Related Activity
TRACE CT and REPLY	• Start, stop, change, or preset the application's tracing options. • Start and stop the external writer. • Connect a trace to and disconnect a trace from an external writer.
DISPLAY TRACE	Obtain status information about traces and external writers.

Note: See *z/OS MVS System Commands* for more information.

IPCS subcommands for component tracing

IPCS Subcommand	Purpose or Related Activity
CTRACE	Format trace entries from a dump or from data sets written to DASD or tape, and display the formatted trace entries.
COPYTRC	Copy trace data from data sets or dumps into one combined data set.
MERGE	Merge trace data from other application traces and GTF trace data, and display the formatted trace entries.

Note: See *z/OS MVS IPCS Commands* for more information.

Exit routines for component tracing

Exit Routine/Model	Purpose or Related Activity
Start/stop exit routine	• Start or stop tracing for the application, or change trace options. • Connect a trace to and disconnect a trace from an external writer.
Display trace exit routine	Provides information about the component trace for the DISPLAY TRACE operator command.
CTRACE buffer find exit routine	Locate trace buffers in a dump for formatting through IPCS.
CTRACE filter/analysis (CTRF) exit routine	Filter trace entries for formatting, or provide data for analysis.
Format models	Provide application-specific format instructions for trace entries.

Note: See *z/OS MVS IPCS Customization* for exits/models related to formatting.

Data areas and mapping macros for component tracing

Data Area/Mapping Macro	Purpose or Related Activity
CTSS (component trace start/stop parameter list) mapped by ITTCTSS	Provide information to the start/stop and display trace exit routines.
CTE (component trace entry) mapped by ITTCTE	Create component trace entries.
ABDPL (ABDUMP parameter list) mapped by BLSABDPL	Provide IPCS information to the CTRACE buffer find exit routine and the CTRF exit routine.
CTXI (component trace exit interface) mapped by ITTCTXI	Provide component trace information to the CTRACE buffer find exit routine and the CTRF exit routine.
ES record (equate symbol record) mapped by BLSRESSY	Record locations of buffers in a dump.
STAB (component trace sublevel table) mapped by ITTSTAB	Define the sublevel table structure pointed to by the component trace start/stop parameter list (CTSSSNTP field).
TBWC (trace buffer writer control area) mapped by ITTTBWC	Communicate between the application and component trace regarding the status of trace buffers. Input for the CTRACECS and CTRACEWR macros.

Note: For more information, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Parmlib members for component tracing

Parmlib Member/Procedure	Purpose or Related Activity
Parmlib member CTncccxx	Use with CTRACE DEFINE or the TRACE CT operator command to establish default or recommended tracing options. The installation is responsible for setting up these parmlib members.
A procedure in SYS1.PROCLIB	Start the component trace writer program (ITTTRCWR) and define the DASD or tape destination for the trace data sets. The installation is responsible for providing this procedure.

Note: See *z/OS MVS Initialization and Tuning Reference* for more information.

When to trace

You have to decide when you want tracing on and, once it is on, what entries to trace. Tracing is **on** when the application is creating some or all of its trace entries. Tracing is **off** when the application is not creating trace entries at any of its trace points. The decision to trace is based on the need to:

- Capture enough data to diagnose a problem.
- Minimize lines of code spent on tracing.
- Optimize the trace path when the trace is on so that tracing causes acceptable performance degradation.

The more trace entries you create, the greater the impact to performance.

Component trace supports **minops** for tracing at a minimum, application-defined default level that cannot be turned off. Some applications choose this option to ensure that some trace data is always available to enable first failure data capture.

Another consideration is whether you can recreate a problem if one arises. If you **can** recreate the problem, you might only turn tracing on after the problem occurs, and then restart the application. If you have to capture data about a problem before it occurs (frequently referred to as first failure data capture) you might choose to trace all the time, or at least at a minimum level. In summary, here are the options you should consider when deciding when to trace:

- Turn the trace on only when a problem occurs.
- Keep the trace on at all times, with the following choices:
  - Trace at a minimum level. (Tracing at a minimum level means creating trace entries at a minimum number of trace points, ignoring the remaining trace points.) This provides some data with a minimal impact to performance.
  - Define trace options to filter which trace entries will be created. By filtering out unneeded trace entries, the trace buffers will contain a higher proportion of useful data.
  - Create all trace entries. In this mode, the application does not support any options to filter the trace data or to turn off the trace. Use this mode if you are tracing a small number of events, or if your first failure data capture requirements outweigh the performance impact.

Where and what to trace

Trace points are the locations in the application's code where the system will create trace entries. You must decide where these trace points will be. Some choices are:

- Module entry
- Module exit
- Start of a functional request
- Major checkpoints in the process of completing a request
- Completion of a functional request
- Interface to another system function
- Interface to the same application on another system
- Any unusual event, such as a detected I/O error or entry to a recovery routine.

You must also decide what information to record in the trace entries, which can hold a variable amount of data. The following are suggestions on the types of data you might place in the trace entries:

- Identification of the unit of work or transaction that is being serviced by the application. This can be the ASID, JOBNAME, USERID, task control block (TCB), service request block (SRB), or transaction identifier.
- For entries that trace the flow of control, the caller's return address.
- For entries that trace the start of a functional request, the input parameters.
- For internal checkpoints, an identification that ties this trace entry to the original request, and information on the current status of the process.
- For unusual events, the cause of the problem and any additional data. For example, you could record a system completion (abend) code and reason code for entry to a recovery routine.
- On return from a service, the return code and reason code (consider having a separate identifier for recording return codes that indicate errors).

- For trace entries being used for analysis rather than as a debugging aid, whatever information the user of the application needs.

Creating trace buffers

You need to obtain storage for:

- The trace buffers, which contain the trace data.
- The TBWC control area, which contains the status information for a specific buffer.
- Control information about your buffers, trace status, and user-defined options.
- An anchor point to control your information.

If you are using an external writer and want to exploit the TESTMODE parameter of the CTRACECS macro, you must initialize the buffer state to available. Either use the CTRACECS macro or set the TBWCCAVL bit to '1'B in the TBWC, mapped by ITTTBWC.

Figure 16 on page 115 includes an illustration of how an application might organize its control structures to manage the component trace buffers.

Where to locate trace buffers: You should anchor your trace buffers in an area that the CTRACE buffer find exit routine can access when running under IPCS. The anchor is a pointer to your control information, and should be an area of storage addressable by your application at all times that the application is defined to component trace. Component trace supports the location of trace buffers in private, common, or data space storage. In z/OS V1R8 or a later release, CTRACE limits the length of buffers that may be passed to the trace writer to X'00000000_80000000' bytes. Substantially smaller buffers are generally recommended. Trace entries within any buffer form a list structure. Loss of one page when a dump is recorded or damage to a single byte can render all remaining trace entries illegible.

When to obtain trace buffers: The application has the choice of allocating the trace buffers at initialization time, or waiting until the trace is turned on. If you wait until the trace is turned on, you avoid using system resources unnecessarily. Either the application or the start/stop exit routine can obtain the needed storage.

Determining the size of your trace buffers: You determine the size of your trace buffers based on the number of trace entries you anticipate creating, and the size of the trace entries. You control both of these values.

You can specify the size of your application's trace buffers in the following ways:

- CTRACE DEFINE with the BUFDFLT parameter
- TRACE CT,nnnnK or TRACE CT,nnnnM operator command
- Parmlib member with BUFSIZE keyword.

If you set a default size on CTRACE DEFINE, the value can be overridden on the TRACE CT command or in a parmlib member, provided you also specify BUFFER=YES on CTRACE DEFINE. If you specify BUFFER=NO, which is the default, you do not allow the buffer size to be changed. In this case, the application might allocate too much storage when it is not needed, or allocate too little storage and cause inadequate trace data to be captured when needed.

Changing the trace buffer size: If the application allows trace buffer size to be changed, the application must deal with the serialization concerns associated with the deletion of the old trace buffers in a multitasking environment. For example, if you decide to use a new, larger buffer, you will have to copy all the data from the old to the new buffer. You must ensure that each task writing trace entries is aware of such things as the new buffer address (and possibly ALET) and the new buffer size, and whether the new buffer is in private, common, or data space storage.

Using multiple buffers: You should allocate multiple trace buffers, especially if you plan to use the external writer. In general, the external writer will asynchronously capture a full trace buffer while the application continues processing and writing trace entries to another trace buffer. If your application does not require trace data to be captured asynchronously, then you might be able to use a single, wrapping trace buffer.

Deciding what to include in the control information: You can set up a block of storage to contain whatever control information you require to keep track of your trace buffers. This control information can be in private, common, or data space storage. The following are examples of the fields you might include in your control information:

Current mode
Whether the trace is currently on or off or min.

Active options
Which tracing options are currently active. Consider using a bit mapping where each bit represents a tracing option. Trace points in the code can check one or more of these bits to determine whether tracing is active for this trace point. See “Setting up user-defined options” on page 96 for further details.

External writer indication
Whether the application is connected to an external writer. If this flag is set, the application issues the CTRACECS macro to keep track of its trace buffers, and the CTRACEWR macro to write buffers to DASD or tape. The application might also store the token that the start/stop exit routine passes. The start/stop routine obtains the token from the CTSSWTKN field of the CTSS (the parameter list passed to the routine). The application needs this token to issue the CTRACEWR macro.

ASID or JOBNAME lists
If the application supports ASID or JOBNAME filtering, the list of ASIDs or JOBNAMEs can be kept here. When filtering, the application can check these lists against the ASID or JOBNAME current at the time of tracing.

Trace buffer addresses
A table, list, or queue of trace buffer addresses and lengths to track the available trace buffers. If the buffers are in different address spaces or data spaces, keep an access list entry token (ALET) or space token (STOKEN) with the buffer pointer.

Trace buffer writer control (TBWC) area
A TBWC is associated with each trace buffer.

Current buffer address
Address of the buffer that is currently receiving trace entries.

Current entry address
Address of the next available slot in the current trace buffer. If your application supports multitasking, consider using compare and swap to serialize the trace buffer and update the pointer to the following trace entry.

Using multiple traces

Component trace allows an application to have more than one trace associated with it. For example, if your application runs in multiple address spaces with multiple tasks in each address space, you could define different traces to represent each address space, and different traces to represent each task in an address space. Applications that have complex logic, with several subtasks, or those that run in multiple address spaces, are good candidates for multiple traces. While multiple traces might appear to use more resources, multiple traces can actually trace more efficiently than single traces, depending on the application.

With multiple traces, called sublevel traces, you can:

- Better control which events are to be traced.
- More easily control and change trace options.
- Capture less frequent trace events, which in single traces might be overwritten by frequent trace events.
- Improve performance because less overhead is required to get addressability to the trace buffers.
- Avoid special serialization of trace buffers, because each trace uses its own trace buffers. Less serialization requires fewer instructions, again contributing to improved performance.
- Set up unique traces with specific options for each function in your application.

For applications that run multiple traces you might want to create special trace entries that can be used to tie the multiple traces together. IPCS supports the merging of multiple traces.

For information about coding the CTRACE macro to set up multiple traces for your application, see *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

Setting up user-defined options

You can categorize your trace entries and control which entries your application will create by setting up user-defined options. The operator then specifies these options on the REPLY operator command, or the options can be in one or more parmlib members. For example, you might have one option for trace entries that identify unusual events, another option for trace entries that trace the flow of control, and other options for trace entries that represent specific functions within the application. You decide what to call these options, and you can set up flags in your control information to correspond to each option.

To understand how to set up and use options, consider the following example:

1. Decide on five options, and name them as follows:
   - ERRORS for unusual events
   - FLOW for events that trace flow of control
   - FUNC1 for events representing "function 1" in your program
   - FUNC2 for events representing "function 2" in your program
   - FUNC3 for events representing "function 3" in your program.
2. Set up flags in your control information to correspond to these options. You might call them FLAG_ERRORS, FLAG_FLOW, FLAG_FUNC1, FLAG_FUNC2, and FLAG_FUNC3.

3. If a problem occurs, you might decide to have the operator turn tracing on with two of the options, ERRORS and FLOW. The operator then indicates ERRORS and FLOW as the options on the REPLY operator command. (You can also do this through a parmlib member.)
4. The system places the options ERRORS and FLOW into the parameter list it passes to the start/stop exit routine.
5. The start/stop exit routine parses the options and sees that ERRORS and FLOW were specified.
6. The start/stop exit routine turns on FLAG_ERRORS and FLAG_FLOW in the control information.
7. The application checks the control information. Because FLAG_ERRORS and FLAG_FLOW are both on, the application creates those trace entries that correspond to options ERRORS and FLOW.

Starting, stopping and changing the trace

Your application can start and stop tracing with or without operator intervention. To start tracing means to start creating the trace entries in the application's code. To stop tracing means to stop creating trace entries, or to revert to the minimum level of tracing. The following are the ways to start, stop and change tracing:

- Through the TRACE CT and REPLY operator commands
- Through a parmlib member:
  - On CTRACE DEFINE
  - On the TRACE CT operator command, with or without PRESET.

See “Using parmlib members” for more information about using parmlib members with component trace.

To start or stop a trace, the system passes control to your start/stop exit routine. Once the exit routine gets control, it can set a flag in the application's control information to indicate that the application should either start or stop tracing.

When you start tracing, either through the operator or through a parmlib member on CTRACE DEFINE, you specify an initial set of trace options. At some point, you might need to change these trace options. If you specify MOD=YES on CTRACE DEFINE, you allow the operator to change the trace options without turning the trace off. If you specify MOD=NO on CTRACE DEFINE (or take the default), the operator must turn the trace off, and then turn the trace back on with the new options.

Using parmlib members

You can use one or more parmlib members to establish default or recommended tracing options for your application. The installation is responsible for creating these parmlib members and placing them in SYS1.PARMLIB. Using parmlib members has several advantages, especially if your application has many trace options associated with it:

- You can avoid operator intervention entirely, or minimize it.
- You can establish tracing options while the application is defining itself to component trace (this could be at a time when operator commands are not yet supported).
- Where operator intervention is necessary, the operator does not have to type in the options, thus avoiding syntax and keystroke errors.

- You can define a set of tracing options for your application before the application is started.

You can specify parmlib members in the following ways:
- On CTRACE DEFINE (can contain information about only a single trace)
- On the TRACE CT operator command (can contain information about more than one trace).

Specifying parmlib member on CTRACE DEFINE macro: When the application issues CTRACE DEFINE with a parmlib member, the system uses the tracing options specified in that parmlib member. If that parmlib member contains instructions to turn the trace on, then the system passes control to the application's start/stop exit routine without any operator intervention. If you update the parmlib member, this changes the tracing options used when the application subsequently issues CTRACE DEFINE with that same parmlib member.

Specifying parmlib member on the TRACE CT operator command: The operator in your installation can specify a parmlib member on the TRACE CT operator command by using the PARM parameter. The operator can even set up tracing options for your application before the application runs and defines itself to component trace.

When the operator issues the TRACE CT command with a parmlib member, the system uses the tracing options specified in that parmlib member, just as it would if you specified the parmlib member on CTRACE DEFINE. However, if you already specified a parmlib member on CTRACE DEFINE, the parmlib member subsequently specified on TRACE CT overrides the parmlib member on CTRACE DEFINE. This override is useful if a problem arises requiring a set of trace options different from those originally specified.

Here is an example of how the operator might specify a parmlib member for application `ABC` on the TRACE CT command, after you specified `CTXABC01` on CTRACE DEFINE:

```
TRACE CT,,COMP=ABC,PARM=CTXABC02
```

The `CTXABC02` parmlib member might contain the following to change the options that were specified in parmlib member `CTXABC01`:

```
TRACEOPTS
ON
ASID(01)
OPTIONS('new options')
```

When you are using multiple traces, a parmlib member that you specify on the TRACE CT command can contain information about more than one head level or sublevel trace. See “Using multiple traces” on page 96 and *z/OS MVS Programming: Authorized Assembler Services Guide* for more information about multiple traces.

Using the PRESET option: When you specify a parmlib member on the TRACE CT command, you can use a parmlib member containing the PRESET option. This allows the operator to set up trace options for your application before starting the application and before the application defines itself to component trace. The operator might do this in anticipation of a problem application. The advantage is immediacy. The operator issues the TRACE CT command with the parmlib member containing the PRESET option. Then, when the operator starts the application, as soon as the application issues CTRACE DEFINE, the system immediately uses the options in the parmlib member specified on TRACE CT.

Here is an example of how the operator might specify a parmlib member with the PRESET option for application ABC:

```
TRACE CT,,COMP=ABC,PARM=CTXABC03
```

The CTXABC03 parmlib member might contain the following to preset the tracing options for application ABC:

```
TRACEOPTS
PRESET(DEFINE)
ON
ASID(01)
OPTIONS('PRESET options...')
```

As soon as the application issues CTRACE DEFINE,NAME=ABC..., the system uses the tracing options in parmlib member CTXABC03. The preset parmlib member overrides the parmlib member that you specify on CTRACE DEFINE.

Externalizing trace data in a dump

You can externalize your trace data in the following types of dumps:

- SVC dump
- Summary dump
- Stand-alone dump.

You can obtain both an SVC dump and a summary dump through the SDUMPX macro. You can obtain a stand-alone dump by asking the operator to request one for you.

Obtaining SVC and summary dumps: To obtain SVC and summary dumps, use the SDUMPX macro to dump the storage containing:

- The trace buffers
- Any control information needed by the CTRACE buffer find exit routine to locate the trace data.

Note: You must specify SDATA=(SQA) on the SDUMPX macro to dump the necessary component trace control information.

If tracing continues while the trace buffers are being dumped, the application can take either of the following steps to ensure useful data in the dump:

- Make the trace buffers large enough so the trace data does not wrap around.
- Suspend tracing until the trace buffers are dumped.

When you use SDUMPX to request an SVC dump, you also get a summary dump by default. Summary dumps can contain up to 4 megabytes (4MB) of data, so the application might plan on receiving about 3.5MB of trace data. If your trace buffers are much larger than 3.5MB, consider suppressing the summary dump and externalizing the trace data through the SVC dump only.

Obtaining a stand-alone dump: To include the trace buffers in any stand-alone dumps, ask the operator to dump the storage containing the trace buffers. You must do one of the following:

- Ensure that the trace buffers and control information are in storage that will always be dumped by the stand-alone dump program (SADMP).
- Set up the SADMP program to specifically dump the desired areas.
- Have the operator specify storage ranges on the prompt from SADMP.

See *z/OS MVS Diagnosis: Tools and Service Aids* for information about the stand-alone dump program (SADMP).

Many components can locate their CTRACE buffers in a dump without the USERDATA supplied by a CTRACE DEFINE operation. IBM advises such components to supply a CTRACE parmlib statement during IPCS session initialization. This allows the CTRACE subcommand to associate the name of the component with the name of the format table supplied by the component. In turn, this permits the component CTRACE buffers to be located and processed regardless of whether a dump contains SDATA=(SQA) information, which is the traditional way that the CTRACE subcommand determines the name of the format table for a component.

Externalizing trace data through the external writer

By using the external writer, you can write application trace buffers directly to a trace data set on DASD or tape rather than requesting a dump. While you might still view your trace buffers by requesting a dump, the advantages of using the external writer are:

- You do not need to code a component trace buffer find exit routine for IPCS processing.
- Depending on the size of the trace data set, you can capture more trace data without using valuable system resources, such as central or auxiliary storage.

The following describes tracing through component trace using an external writer.

- The programmer plans and sets up the user-defined control structures.
- The installation establishes a procedure for the external writer in SYS1.PROCLIB.
- The application defines itself to component trace with an external writer.
- The operator starts the external writer.
- The operator turns on the application's trace and connects the application to the external writer.
- The system passes control to the start/stop exit routine to start tracing.
- The application starts tracing by creating trace entries and placing them in trace buffers.
- The application manages trace buffers using the external writer.
- The operator turns off the application's trace; this action disconnects the application from the external writer.
- The system passes control to the start/stop exit routine to stop tracing.
- The operator stops the external writer.
- The application deletes itself from component trace.
- A diagnostician or systems analyst formats the trace data sets through IPCS.

Using an external writer involves a number of trace-related activities. The trace-related activities are:

Trace-related activity:	**How to accomplish the activity:**
Define the application to component trace allowing connection to an external writer	CTRACE DEFINE with the WTR and WTRMODE parameters

Trace-related activity:	**How to accomplish the activity:**
Start an external writer	An installation-written procedure in SYS1.PROCLIB and one of the following: • TRACE CT operator command with the WTRSTART parameter • On a TRACE CT operator command, a parmlib member with PRESET and WTRSTART parameters • On a TRACE CT operator command, a parmlib member with the WTRSTART parameter • On CTRACE DEFINE, a parmlib member with the WTRSTART parameter
Connect an external writer	An installation-written procedure in SYS1.PROCLIB and one of the following: • TRACE CT,ON operator command and REPLY operator command with the WTR=*wtrname* parameter • On a TRACE CT operator command, a parmlib member with PRESET and WTR parameters • On a TRACE CT operator command, a parmlib member with the WTR parameter • On CTRACE DEFINE, a parmlib member with the WTR parameter
Manage trace buffers	• CTRACECS macro • CTRACEWR macro • ITTTBWC mapping macro
Disconnect an external writer	One of the following: • TRACE CT,OFF operator command • TRACE CT,ON operator command and REPLY operator command with the WTR=DISCONNECT parameter • On a TRACE CT operator command, a parmlib member with the WTR=DISCONNECT parameter
Stop an external writer	One of the following: • TRACE CT operator command with the WTRSTOP parameter • On a TRACE CT operator command, a parmlib member with the WTRSTOP parameter

Starting the external writer Starting an external writer means preparing the external writer to receive requests and write trace buffers out to trace data sets. To start an external writer, the operator issues the TRACE CT command with the WTRSTART parameter. An external writer that has been started is said to be active.

To start an external writer and connect an application to it without operator intervention, specify a CTncccxx parmlib member on the PARM parameter of CTRACE DEFINE.

Connecting an application to the external writer: Connecting an application to an external writer means the application can write out its trace buffers to an active external writer. To connect to an external writer, the operator issues the TRACE CT and REPLY operator commands. To connect an external writer without operator intervention, you can specify the CTncccxx parmlib member on the PARM parameter of CTRACE DEFINE. For example, the application can issue:

```
CTRACE DEFINE,NAME=ABCD,WTR=YES,PARM=CTXABC04
```

Parmlib member CTXABC04 might look like this:

```
TRACEOPTS
WTRSTART(CTWDASD)
ON
WTR(CTWDASD)
ASID(ASID01)
OPTIONS ('option1,option2')
```

When you start and connect the component trace with an external writer, the system passes control to your start/stop exit routine. Once the start/stop exit gets control, it should set a flag in the application's control information to indicate that the application should start managing its trace buffers to be passed to the external writer.

Disconnecting an application from the external writer: Disconnecting an application from an external writer means that the application cannot write out its trace buffers to an active external writer. To disconnect an external writer, the operator issues the TRACE CT and REPLY commands.

Note: Turning the application's trace off causes an automatic disconnect of that application from the external writer. In normal processing there is no need to disconnect an application from an external writer. Simply turning the trace off will automatically cause a disconnect to occur.

When you disconnect the application from the external writer or stop the trace, the system passes control to your start/stop exit routine. Once the start/stop exit gets control, it should set a flag in the application's control information to indicate that the application no longer needs to manage trace buffers to be passed to the external writer.

Stopping the external writer: Stopping an external writer means any application trace requests sent to the external writer will not be written out to trace data sets. To stop an external writer, the operator issues the TRACE command with the WTRSTOP parameter.

If the application supports the external writer and either dynamically changes the buffer size or deletes buffers in a start/stop routine, then there may be a data integrity exposure when minimum options (MINOPS) is not supported. Component trace could possibly copy storage that is now allocated to another task because the buffer was deleted before copying was complete. Applications that fall into this situation could use TESTMODE=AVAIL to check the buffer's availability before deleting it, or not change the buffer size when the external writer is active. There is no integrity problem if MINOPS is supported and the buffer size does not change.

If you are running under z/OS V1R8 or a later release, you may place your buffers above the bar. Each buffer can be as long as X'FFFFFFFF_80000000' bytes. Individual CTRACE entries remain limited to 64000 bytes in length.

Coding macros for application traces

This information describes the coding for CTRACE, CTRACECS and CTRACEWR macros. The CTRACE macro is needed to define the application to component trace and delete the application from component trace. CTRACECS manages trace buffers and CTRACEWR writes the buffers to an external data set. The following is a high level sequence of events for your application's tracing with component trace.

Sequence of events for trace:

1. The application issues CTRACE DEFINE macro during initialization.
2. The application's start/stop routine receives control when the operator turns the trace on.
   - The routine obtains trace buffers and TBWC (Trace Buffer Writer Control) block of storage.
   - Each buffer has an associated TBWC control area to store the state of the buffer and sequence number associated with the buffer. A unique buffer sequence number must be maintained for every CTRACEWR macro invocation.
   - The routine initializes the TBWC for each buffer to the available state.
3. The application creates trace entries according to options.
   - If component trace is active, it finds an available buffer by looping through TBWCs, issuing the CTRACECS macro to put the buffer in the filling state if the buffer is currently in the available state, until one is marked filling.

     The CTRACECS macro serializes changing the state and sequence number in the TBWC.
   - The application fills the buffer with trace data, and issues CTRACECS macro to put the buffer in the full state if the buffer is currently in the filling state.
   - If the external writer is active, the application issues the CTRACEWR macro to write the buffer to the external data set. Otherwise, if buffers are being wrapped in storage, the application issues the CTRACECS macro to mark the buffer available.
4. During SDUMP or standalone dump, dump component buffers.
5. The application provides a locate trace buffer exit for dumps.
6. The application provides a unique filter exit.
7. The IPCS CTRACE subcommand provides common functions like time stamp, ASID, and JOBNAME filtering.
8. Each trace entry that passes all of the filters is formatted according to the application's format table definition.

Using the CTRACE macro to define the application to component trace

To use component trace, you must first define the application to component trace by issuing the CTRACE macro with the DEFINE parameter (CTRACE DEFINE). CTRACE DEFINE defines the application as being available for activities such as starting and stopping a trace, displaying trace information, and formatting trace data. Through parameters on CTRACE DEFINE, you specify information about the application, such as:

- The application name (**NAME** parameter). The operator uses this name on the TRACE CT command (COMP parameter). When formatting trace data with IPCS, use this name on the IPCS CTRACE subcommand (COMP parameter). The application name is required.

  Note: *Do not* use names that begin with the prefix *SYS*. The prefix *SYS* is reserved for use by IBM.
- A parmlib member name (**PARM** parameter). Identifying a parmlib member establishes the initial tracing options for an application. These options can be overridden by operator commands. This parameter is optional. (See “Using parmlib members” on page 97 for more information.)
- A head level (**HEAD** parameter) and a sublevel trace name (**SUB** parameter). These parameters are related to multiple traces. (See “Using multiple traces” on page 96 for more information).
- The load module that contains the CTRACE format table (**FMTTAB** parameter), which controls trace formatting in the IPCS CTRACE subcommand.
- The name of the start/stop exit routine (**STARTNAM** parameter). The system gives this routine control when the operator enters a TRACE CT command or when you specify the PARM parameter on CTRACE DEFINE.
- The name of the display trace exit routine (**DISPNAM** parameter). The system gives this routine control when the operator enters a DISPLAY TRACE command.
- A list of default, user-defined tracing options (**MINOPS** parameter). If you specify MINOPS, tracing with these options will be in effect at all times and cannot be turned off.
- An indication that the application supports an external writer (**WTR** and **WTRMODE** parameters).

Specifying parmlib on CTRACE DEFINE: When you specify a parmlib member on CTRACE DEFINE, the system uses the tracing options specified in that parmlib member. If that parmlib member contains instructions to turn the trace on, then the system passes control to the application's start/stop exit routine without operator intervention. If you update the parmlib member, you change the tracing options used when the application subsequently issues CTRACE DEFINE with that same parmlib member.

Here is an example of how you might specify a parmlib member for application `ABC` on CTRACE DEFINE:

```
CTRACE DEFINE,NAME=ABC,PARM=CTXABC01...
```

The parmlib member `CTXABC01` might contain the following to turn tracing on for address space 01 with specific options:

```
TRACEOPTS
ON
ASID(01)
OPTIONS('original options')
```

The `CTXABC01` parmlib member takes the place of the following operator commands:

```
TRACE CT,ON,COMP=ABC
REPLY xx,ASID=(01),OPTIONS=('original options'),END
```

A parmlib member you specify on CTRACE DEFINE can contain information about only a single trace. If you are defining multiple traces, include in each parmlib member options that pertain only to the specific head level or sublevel trace. The following topic explains head level and sublevel traces.

Using multiple traces: To use multiple traces, you must create a head level and sublevel traces. A head level is a control level with sublevel traces associated with it. Figure 11 uses an inverted tree structure to depict the relationship of a head level and sublevel traces for application APPLABC.

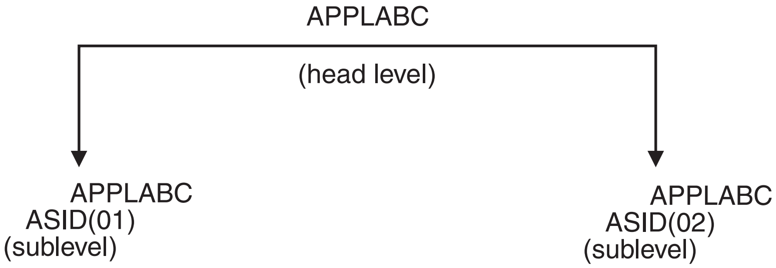


Figure 11. Structure of Multiple Traces for an Application

Creating a head level: A head level is a control level set up to support sublevel traces, and can itself be a trace. Create a head level by specifying HEAD=YES on CTRACE DEFINE. If you want sublevel traces to be able to share the attributes, options, and mode of the head level, specify both of the following:

- HEADOPTS=YES on CTRACE DEFINE when you create the head level
- LIKEHEAD=YES on CTRACE DEFINE when you create the sublevel traces.

Creating sublevel traces: Create a sublevel trace by specifying SUB=*subname* on CTRACE DEFINE. Obtain separate trace buffers for each sublevel trace. You can define sublevel traces as either:

- Independent traces with unique tracing features (LIKEHEAD=NO on CTRACE DEFINE)
- Traces that use a head level's attributes, options and mode (LIKEHEAD=YES on CTRACE DEFINE).

Figure 12 on page 106 lists the characteristics, options, and status associated with a trace. If an application defines a sublevel trace to match its head level, LIKEHEAD, any changes to the head level options and status are reflected in the sublevel trace.

Turning a sublevel trace on or off or changing its options cancels its LIKEHEAD status. A status of LIKEHEAD simply means that the sublevel trace has the same status and options as its head level. When a sublevel trace has its LIKEHEAD status cancelled, subsequent changes to its head level do not affect the sublevel trace except when the head level is turned off. When a head level trace is turned off, sublevel traces that were defined with LIKEHEAD=YES on the CTRACE DEFINE macro will also be turned off, regardless of their LIKEHEAD status.

Attributes (parameters on CTRACE macro)

- ASIDS=YES/NO
- JOBS=YES/NO
- MINOPS
- MOD=YES/NO
- BUFFER=YES/NO
- BUFMIN
- BUFMAX
- BUFDFLT
- BUFDEFIN=YES/NO
- WTR=YES/NO

Characteristics (parameters on CTRACE macro)

- STARTNAM
- FMTTAB
- HEAD=YES/NO
- LIKEHEAD=YES/NO
- MANYSUBS=YES/NO

Options (parameters on TRACE CT operator command or in a parmlib member)

- ASID
- JOBNAME
- OPTIONS
- BUFSIZE
- WTR

Status (parameters on TRACE CT operator command or in a parmlib member)

- ON
- OFF

Figure 12. Trace Features

Note: When you specify attributes and characteristics on CTRACE DEFINE, these features do not change. When you specify options and status through the TRACE CT operator command or in a parmlib member, you can change these features dynamically.

A sublevel trace can also serve as a head level. Specify HEAD=YES and SUB=*subname* to define a sublevel that is also a head level. In this case, all of the sublevel and head level attributes, options, and status apply to the subsequent sublevels if they are likehead. In Figure 13 on page 107, ASID(01) is a sublevel trace of APPLABC, and a head level for FUNCTION5 and FUNCTION7. FUNCTION5 and FUNCTION7 use the attributes, options and status of both ASID(01) and APPLABC.

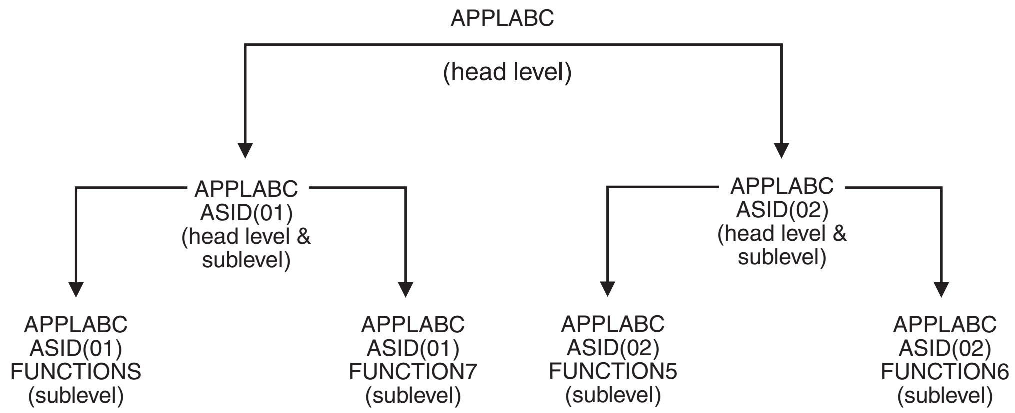


Figure 13. Structure of Multiple Traces for an Application

The inverted tree structure illustrates how an application named APPLABC supports separate traces for each user, and has multiple functions in use in the same address space. In addition, the same function (FUNCTION5) is active in two address spaces. A description of the steps in defining the tree structure follows:

1. During initialization, APPLABC issues CTRACE DEFINE and specifies APPLABC as the head level (by specifying HEAD=YES and NAME=APPLABC on CTRACE DEFINE). APPLABC also specifies HEADOPTS=YES to allow its sublevel traces to share its attributes, options, and status.
2. The first time APPLABC gets control in address space ASID(01), APPLABC issues CTRACE DEFINE for sublevel ASID(01), with LIKEHEAD=YES **and** HEAD=YES. Now, ASID(01) is established as a sublevel trace with the same tracing attributes, options, and status as its head level, APPLABC, and ASID(01) is also established as a head level.
3. APPLABC issues another CTRACE DEFINE for sublevel trace FUNCTION5, with LIKEHEAD=YES. Now FUNCTION5 is established as a sublevel trace, and uses all the attributes, options, and status from its head level, ASID(01), and ASID(01)'s head level, APPLABC.
4. APPLABC defines FUNCTION7 in the same way as FUNCTION5.
5. When APPLABC gets control in address space ASID(02), APPLABC issues CTRACE DEFINE to establish itself as a head level, and again to establish ASID(02) as a sublevel trace with the same parameters as ASID(01).
6. APPLABC establishes FUNCTION5 and FUNCTION6 as sublevel traces. Even though FUNCTION5 is also in ASID(01), FUNCTION5 can have unique tracing features in ASID(02) by specifying LIKEHEAD=NO (or taking the default).

Figure 14 on page 108 illustrates how you would issue CTRACE DEFINE to set up the multiple traces for `APPLABC`.

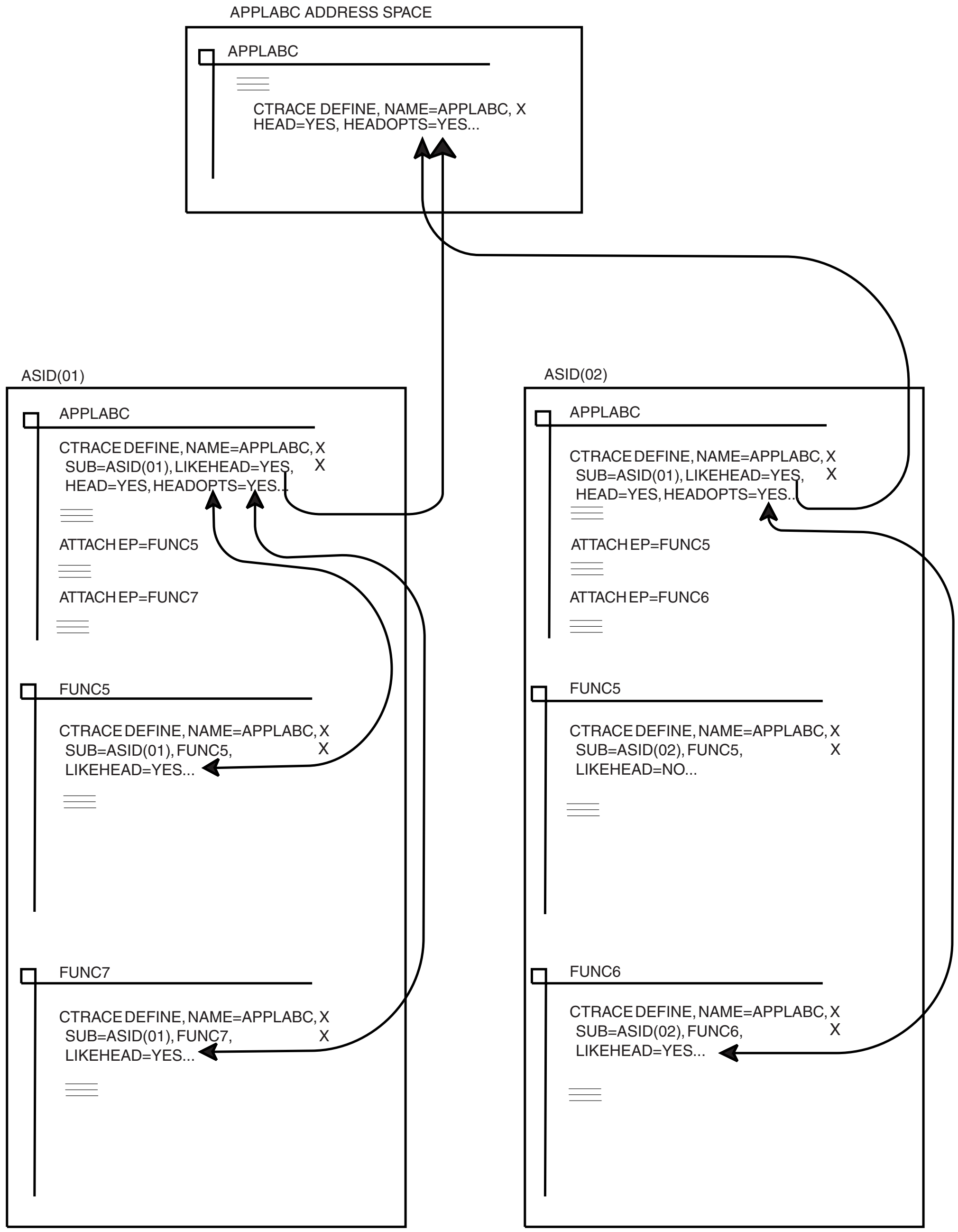

APPLABC ADDRESS SPACE
APPLABC
CTRACE DEFINE, NAME=APPLABC, X
HEAD=YES, HEADOPTS=YES...
ASID(01)
APPLABC
CTRACE DEFINE, NAME=APPLABC, X
SUB=ASID(01), LIKEHEAD=YES, X
HEAD=YES, HEADOPTS=YES...
ATTACH EP=FUNC5
ATTACH EP=FUNC7
FUNC5
CTRACE DEFINE, NAME=APPLABC, X
SUB=ASID(01), FUNC5, X
LIKEHEAD=YES...
FUNC7
CTRACE DEFINE, NAME=APPLABC, X
SUB=ASID(01), FUNC7, X
LIKEHEAD=YES...
ASID(02)
APPLABC
CTRACE DEFINE, NAME=APPLABC, X
SUB=ASID(01), LIKEHEAD=YES, X
HEAD=YES, HEADOPTS=YES...
ATTACH EP=FUNC5
ATTACH EP=FUNC6
FUNC5
CTRACE DEFINE, NAME=APPLABC, X
SUB=ASID(02), FUNC5, X
LIKEHEAD=NO...
FUNC6
CTRACE DEFINE, NAME=APPLABC, X
SUB=ASID(02), FUNC6, X
LIKEHEAD=YES...

Figure 14. Setting up Multiple Traces with CTRACE DEFINE

Using the component trace external writer: To define your application to component trace to support an external writer, issue CTRACE DEFINE with the WTR=YES and WTRMODE parameters. The WTR=YES parameter indicates the application supports having its trace buffers written to trace data sets on DASD or tape. The WTRMODE parameter specifies the type of storage your application is using for its trace buffers. The WTRMODE values are PAGEABLE, DREF, or FIXED.

Note: IBM recommends you keep your trace buffers in pageable storage, so you do not deplete your system's central storage. However, if your application cannot accept page faults, you should use either disabled reference (DREF) or fixed storage.

Using CTRACECS to manage trace buffer status

Your application is responsible for managing trace buffers when writing to them and deleting them. Component trace manages the buffers while they are being captured until they are ready for reuse. Your application uses the CTRACECS macro to update and track the status of its buffers. Each buffer has an associated trace buffer writer control (TBWC) area to store the state of the buffer and the sequence number associated with the buffer. Each buffer has 4 possible states:

State	Description	Set by:
FILLING	trace is filling the buffer	application
FULL	full and ready to be written to datasets	application
CAPTURE	Component trace is writing to datasets	Component trace
AVAILABLE	available for use by the trace	Component trace

CTRACECS, which sets the fields in the TBWC area mapped by the ITTTBWC mapping macro, tests the state of the application's trace buffers to prevent data loss due to buffers being overwritten before being written to tape or DASD by the CTRACEWR macro. The current state of the application's trace buffer can be compared to the expected buffer state, before setting the buffer to the requested state. You can use the CTRACECS macro to ensure that:

- Only when the buffer is in the available state can it be set to filling.
- Only when the buffer is in the filling state can it be set to full.
- Only when the buffer is in the full state can it be set to available. When the writer is active there is no need to use the CTRACECS macro to mark the buffer available.
- You are working with the correct buffer by comparing the current buffer sequence number to the expected buffer sequence number.

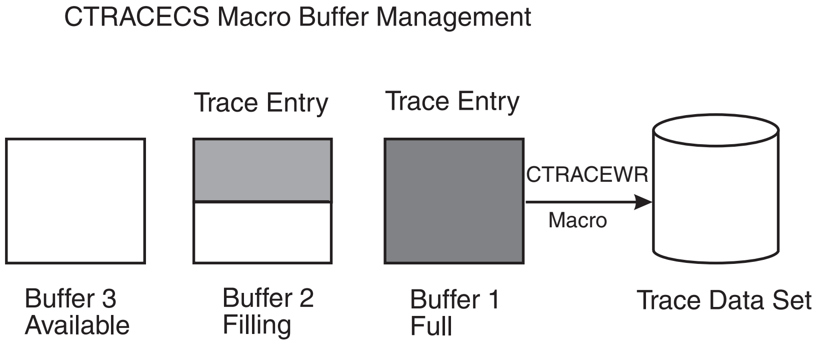


Figure 15. Managing Buffer Status

The following steps are an example of how you might use the CTRACECS macro to manage multiple trace buffers:

1. Mark all buffers available. You cannot use the TESTMODE keyword in subsequent invocations if the buffer is not initialized to a known state.

```
CTRACECS   TBWC=TBWCAREA,                                    x
           MODE=AVAIL,                                       x
           COM='Initialize buffers to available state'
```

2. Mark an available buffer filling and start putting data into it.

```
              LA R6,3                                      x
              ST R6,EXPSEQ                                 x

  CTRACECS    TBWC=TBWCAREA,                               x
              TESTMODE=AVAIL,                              x
              MODE=FILLING,                                x
              CSLABEL=NOT_AVAIL,                           x
              BUFFSEQ#=NEWSEQ#,                            x
              COM='Mark Buffer filling if available'
              .
              .
NOT_AVAIL
*The buffer is not available, try another buffer.
```

3. When there is no more room for trace entries in the buffer, mark the buffer full.

```
            LA R6,2                                      x
            ST R6,EXPSEQ                                 x
                                                         x
  CTRACECS  TBWC=TBWCAREA,                               x
            MODE=FULL,                                   x
            TESTMODE=FILLING,                            x
            TESTSEQ#=EXPSEQ,                             x
            CSLABEL=NOT_FILLING,                         x
            COM='MARK buffer full if filling'
            .
            .
NOT_FILLING
*The buffer is not filling or the buffer sequence number
assigned to the buffer is not the expected value.  The
buffer is not marked full.
```

4. If using the external writer, invoke the CTRACEWR macro to write the trace buffer entries to the external data set. Component trace will mark the buffer available, once CTRACEWR has finished writing the trace entries to the data set.

   If you are not using an external writer, mark the buffer available yourself so that it can be reused.

```
           LA R6,1                                       x
           ST R6,EXPSEQ                                  x
                                                         x
 CTRACECS  TBWC=TBWCAREA,                                x
           MODE=AVAIL,                                   x
           TESTSEQ#=EXPSEQ,                              x
           CSLABEL=WRONGBUFFER,                          x
           COM='Mark buffer# 1 available'
           .
           .
 WRONG_BUFFER
 *The is not buffer# 1 and is not marked available.
```

The TBWC mode field indicates the requested status of the trace buffers.

MODE Status

AVAIL
The trace buffer is available and ready to be filled with trace entries.

FILLING
The application is entering trace entries into the trace buffer.

FULL The application cannot enter any more trace entries into the trace buffer.

CAPTURE
The system is writing the trace buffer out to DASD or tape.

The CTRACECS macro requires that a unique TBWC buffer sequence number be provided by the application. The application must serialize the buffer sequence number assigned to each buffer. The sequence number must be unique for every buffer passed to the external writer on the CTRACEWR macro.

Support for testing the application's trace buffers is provided by the TESTMODE and TESTSEQ# parameters. The system updates the TBWC to the requested status, if:

- The expected status (TESTMODE) is equal to the current mode (MODE), and
- The expected buffer sequence number (TESTSEQ#) equals the current buffer sequence number (BUFFSEQ#).

A successful comparison will update the TBWC to the requested status. Otherwise, the application will branch to a subroutine that you specified using the CSLABEL keyword.

CSLABEL is optional with TESTMODE=CURRENT but required with TESTMODE=AVAIL/FULL/FILLING and TESTSEQ#. If you specify CSLABEL=RETRY, the default, the code will branch to a generated label that retries the current value of the TBWC until the update is successful.

Note: IBM recommends using TESTMODE and TESTSEQ# to test the state of the buffer.

The following lists the keyword parameters for TESTMODE:

TESTMODE
 Status

AVAIL
 Requests that the state of the buffer be set to the state requested by the MODE keyword only when the buffer is in the available state.

FILLING
 Requests that the state of the buffer be set to the state requested by the MODE keyword only when the buffer is in the filling state,

FULL Requests that the state of the buffer be set to the state requested by the MODE keyword only when the buffer is in the full state.

CURRENT
 Is the default value that sets the state of the buffer to the state specified by the MODE keyword regardless of the current state.

The TBWC contains the current buffer sequence number (TBWCSEQ), which must also equal the expected buffer sequence number (TESTSEQ#), before it can be updated to the desired buffer status. Use TESTSEQ# to ensure a new buffer sequence number is not being associated with the current TBWC. All four TBWC fields and the buffer sequence number are used to communicate between your application and a component trace external writer.

Your application might do the following:

- Check whether a trace buffer is available to be filled using TESTMODE=AVAIL. IBM recommends more than one trace buffer when tracing with an external writer and of course the TBWC would need to be initialized to the available state when it is obtained.
- If no buffers are available, your application should consider obtaining more buffers or increase the buffer size the next time the trace is started.

- When using the CTRACECS macro to mark a new buffer FILLING, make sure that the buffer sequence number is always incremented. The buffer sequence number is in the TBWC. Increment the buffer sequence number when using the external writer even if the return code from the CTRACEWR macro is not zero.

THE CTRACEWR macro: To write trace buffers to trace data sets on DASD or tape, your application must be connected to an external writer. The CTRACEWR macro writes the trace buffers out to trace data sets. The trace buffer status should be set to full by the application using the CTRACECS macro before issuing the CTRACEWR macro. When the CTRACEWR macro is issued, CTRACE marks the buffer captured and when CTRACE is finished capturing the buffer, it marks the buffer available for use again. If the application tries to reuse the buffer when it is marked captured, CTRACE will not write the buffer to the external data set.

Deleting the application from component trace

You can delete an application from component trace by issuing the CTRACE macro with the DELETE parameter (CTRACE DELETE). After a CTRACE DELETE, component trace no longer honors TRACE CT operator commands against that trace. If you issue the CTRACE DELETE before the trace buffers are dumped, component trace no longer honors IPCS CTRACE subcommands against that application.

You should delete the application as part of the normal cleanup of resources. In the event of abnormal termination, you should include CTRACE DELETE as part of recovery or termination cleanup to avoid problems when you restart the application. Deleting the application from component trace prevents the system from displaying an inactive trace as active on the DISPLAY TRACE operator command.

Applications that use multiple traces should be careful when deleting their head level. Deleting a head level results in the deletion of all sublevel traces associated with that head level. See “Using multiple traces” on page 96 for more information on multiple traces.

Coding a start/stop exit routine

An application that will be using component trace must have a start/stop exit routine that starts and stops tracing for the application or modifies trace options. You are responsible for writing this routine.

If your CTRACE DEFINE macros define multiple traces for your application, you must write a start/stop exit routine for each trace.

Exit routine environment

The start/stop exit routine receives control in the following environment:

Environmental factor	Requirement
Authorization:	Supervisor state and PSW key 0
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN = SASN
AMODE:	31-bit. All passed areas can be above 16 megabytes in virtual storage.
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held

Environmental factor	Requirement
Serialization:	Component trace prevents multiple conflicting commands from being processed simultaneously.
Location:	The start/stop routine must be in LPA or the LNKLST concatenation. If the start/stop routine is in LPA, the system passes control to the module at that location, and the module is not page-fixed. Otherwise, the system places the start/stop routine into the fixed common service area (CSA).

Exit Recovery: The start/stop routine must supply its own ESTAE-type recovery routine or functional recovery routine (FRR) for any needed recovery or cleanup of resources.

Exit routine processing

The start/stop routine receives control to start or stop tracing for the application, or to change trace options. The routine can determine what action to take by checking bits in the CTSSFLGS field of the component trace parameter list (CTSS), which is passed to the start/stop routine. If the CTSSSTRT bit is on, the request is to start tracing. If the CTSSSTOP bit is on, the request is to stop tracing.

When the start/stop routine receives control to start tracing for the application or to change the trace options, the routine can do the following:

- Parse the options in the OPTIONS parameter passed to the routine (either from the REPLY operator command or a CT*ncccxx* parmlib member). If the routine detects errors, it can issue a WTO using the console ID and CART specified in the CTSS, and set a failing return code.
- Check for other errors.
- Obtain storage for trace buffers in common storage, in the private area, or in a data space. IBM recommends that you allocate multiple buffers. You might have to schedule an SRB, post a task, or switch the primary address space to get to the right mode to obtain storage.
- If any problems occur locating the desired address space or obtaining storage, set the appropriate return and reason codes.
- If you plan to externalize your data through a dump data set, anchor the trace buffers in an area that the CTRACE buffer find exit routine can access when running under IPCS. See *z/OS MVS IPCS Customization* for information about writing this routine.
- Use the 16 bytes of user data in the CTSSUSRD field of the CTSS to assist in locating the trace buffers. The application can specify this user data on CTRACE DEFINE with the USERDATA parameter, or in the CTSSUSRD field when the start/stop routine gets control.
- Set a bit (CTSSRNSS) in the CTSS to indicate whether the start/stop routine for sublevel traces defined with the LIKEHEAD keyword should be called individually by component trace.
- Save ASIDs, JOBNAMEs, and the results of the OPTIONS parse in storage where the application can access them during its tracing activity.
- If the routine receives control indicating connection to an external writer (the CTSSWCON flag is on), pass the token contained in the CTSSWTKN field to the application. The application needs this token when it issues the CTRACEWR macro to write buffers to DASD or tape. See “Externalizing trace data through the external writer” on page 100 for information about the external writer.

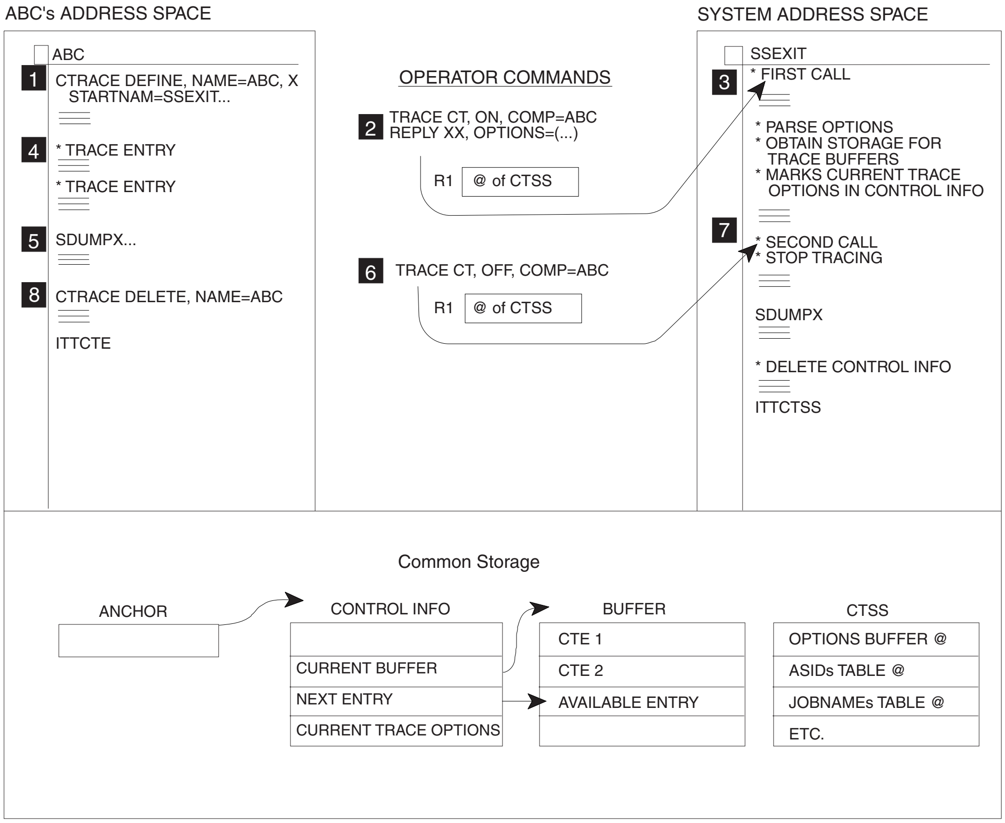
For multiple traces in an application, each routine receives a unique token, whether the CTRACE DEFINE macro contains LIKEHEAD=NO or LIKEHEAD=YES. Specifying these unique tokens in the CTRACEWR macros differentiates the buffers for the head level and sublevel trace entries. You can use IPCS to view a sublevel's entries from one of its buffers or, with the IPCS MERGE subcommand, to group a sublevel's buffers and view the grouped entries. any likehead traces.

- When deleting a trace with CTRACE DELETE, the head level start/stop routine will get control with the CTSSRNSS bit on, indicating that as a default, the start/stop routine for any likehead traces will also run. If you do not want these routines to run, code the head level start/stop routine to turn the CTSSRNSS bit off.
- When changing a trace, if the routine receives control indicating disconnection from an external writer (the CTSSWDIS flag is on), determine if the application has any more buffers to write. If so, use the CTRACEWR macro with the SYNCH=YES keyword to write the current buffer to the external dataset. See "Externalizing trace data through the external writer" on page 100 and "Using multiple traces" on page 96 for further information.
- If the routine detects no errors, set the return code to zero and perform the requested action to turn on the application's trace or change options.
- Establish the returned general purpose registers. See "Return specifications" on page 120 for the return specifications.
- Return control to component trace.

When the start/stop routine receives control to stop tracing for the application, the routine should do the following:

- Stop tracing or revert to the minimum level of tracing. If the application specified MINOPS on CTRACE DEFINE, the tracing should revert to the minimum level as defined by the application.
- If the application trace is also connected to an external writer, then the CTSSWDIS bit will also be on, indicating that the application must also disconnect from the external writer. Use the CTRACEWR macro with the SYNCH=YES keyword to write the current buffer to the external dataset. See "Externalizing trace data through the external writer" on page 100 and "Using multiple traces" on page 96 for further information.
- Set the return code in general purpose register (GPR) 15. See "Return code specifications" on page 124 for the return code specifications.
- Return control to component trace.

Figure 16 on page 115 illustrates the interaction between the application, its start/stop exit routine, and the operator.



1 Application ABC defines itself to component trace.

2 A problem occurs, so the operator turns on the trace.

3 The system passes control to the start/stop exit routine to start tracing for the application.

4 ABC starts creating trace entries, based on user-defined options.

5 When done creating trace entries, ABC issues the SDUMPX macro to dump the storage containing the trace buffers.

6 The operator turns the trace off.

7 The system passes control to the start/stop exit routine to stop tracing for the application, and optionally dump the trace buffers.

8 ABC deletes itself from component trace.

Figure 16. Tracing with Component Trace. Interaction between the application, its start/stop exit routine, and the operator.

Programming considerations

Consider the following when writing the start/stop exit routine:

- If you set a non-zero return code, component trace assumes one of the following:
  - The entire TRACE CT command failed. Component trace discards all of the command.
  - The entire CTRACE DEFINE failed. Component trace discards all of the macro.

  A DISPLAY TRACE command will show the previous status.
- If you use a parmlib member on CTRACE DEFINE to start the trace, and you set a non-zero return code, then the define will fail.

- If the start/stop exit routine detects an error and does not set a failing return code, this will result in incorrect trace options being displayed through the DISPLAY TRACE operator command.
- A TRACE CT operator command always results in the start/stop exit routine getting control. However, determining whether the start/stop exit will get control during CTRACE DEFINE is more complex when you are using multiple traces or parmlib members. In general, a change in the *status* of the trace results in the start/stop exit routine getting control.

  The *status* of the trace consists of the trace mode, buffer size, and options. When you are using multiple traces, a status of **likehead** means that the status of a sublevel trace is the same as the status of its head level.

  Table 18 lists possible combinations of mode and parameters and, based on the combinations, tells whether the start/stop routine runs. See "Using multiple traces" on page 96 for more detailed information about multiple traces.

  Note:

  1. On CTRACE DEFINE, LIKEHEAD=NO is the default. In the column labeled **Parameters on CTRACE Macro**, assume LIKEHEAD=NO unless specified.
  2. **Parameters in Parmlib Member** gives the parameters in a parmlib member only if you specified one on CTRACE DEFINE; otherwise, this column is blank.
  3. **PRESET Status** means that the parmlib member specified on the TRACE CT command has the PRESET option. If PRESET is not in effect, this column is blank.
  4. **Mode of Head Level** is relevant only when the trace is **likehead**. If **likehead** is not in effect, this column is blank.
  5. The figure does not show CTRACE DEFINE with both the LIKEHEAD=YES and PARM=*parm* parameters because this combination results in an error.

Table 18. Conditions Determining When Start/Stop Routine Runs

Parameters on CTRACE Macro	Parameters in Parmlib Member	PRESET Status	Mode of Head Level	Will Start/Stop Routine Run?	Comments
DEFINE...				No	
DEFINE, PARM =*parm*...	ON			Yes	
DEFINE, PARM =*parm*...	OFF			No	
DEFINE, PARM =*parm*...	LIKEHEAD		ON	Yes	
DEFINE, PARM =*parm*...	LIKEHEAD		OFF	Yes	Specification of LIKEHEAD results in the exit getting control, even though the mode of the head level is **off**.
DEFINE...		ON		Yes	
DEFINE...		OFF		No	
DEFINE...		LIKEHEAD	ON	Yes	

Table 18. Conditions Determining When Start/Stop Routine Runs (continued)

Parameters on CTRACE Macro	Parameters in Parmlib Member	PRESET Status	Mode of Head Level	Will Start/Stop Routine Run?	Comments
DEFINE...		LIKEHEAD	OFF	Yes	Specification of LIKEHEAD results in the exit getting control, even though the mode of the head level is **off**.
DEFINE, LIKEHEAD =YES...			ON	Yes	
DEFINE, LIKEHEAD =YES...			OFF	No	
DEFINE, LIKEHEAD =YES...		ON		Yes	The mode of the head level is not relevant because the preset changed the trace from **likehead** to **on**.
DEFINE, LIKEHEAD =YES...		OFF		Yes	The mode of the head level is not relevant because the preset changed the trace from **LIKEHEAD** to **off**. The change results in the exit getting control.
DEFINE, LIKEHEAD =YES...		LIKEHEAD	ON	Yes	
DEFINE, LIKEHEAD =YES...		LIKEHEAD	OFF	No	

Start/stop exit routine communications

The start/stop exit routine runs either in a system address space or the address space of the caller of CTRACE DEFINE. When a TRACE CT operator command causes the start/stop exit to run, the routine runs in a system address space. If a CTRACE DEFINE causes the start/stop exit to run, either through a parmlib member specified on CTRACE DEFINE or through a parmlib member with the PRESET option, the routine runs in the caller's address space.

In either case, the routine might have to communicate with another address space to make the trace options effective. The following methods to accomplish this are described in this information:

- Use commonly addressable storage:
  - Address space common areas
  - SCOPE=ALL data spaces
  - SCOPE=COMMON data spaces.
- Use a program call (PC) routine
- Schedule a service request block (SRB).

Using commonly addressable storage: If the application doing the tracing does not have its own address space, then it can maintain its trace options in commonly addressable storage. The start/stop routine can copy the desired options from the CTSS to the application-maintained trace control area in commonly addressable storage.

Applications that do have their own address space can attach a permanent task to process trace requests. When the start/stop exit routine gets control, it could do the following:

- Obtain common storage to pass the CTSS information. (Consider setting up a permanent area to hold the CTSS information.)
- Copy the CTSS information into this area.
- Post the task that processes the trace requests. (The ECB must be in common storage.)
- Wait for the task to complete.
- Upon being posted with a return code, send the return code back to component trace in register 15.

The task that processes the trace requests could do the following:

- Wait for a trace request.
- When posted, move the trace options to private storage.
- Parse any application-unique options if not already done in the start/stop exit routine.
- Process the request to start, stop, or change the trace options.
- Post the start/stop exit with a return code.
- Loop back to wait for the next request.

Using a PC routine: The application can implement a space-switching PC to get addressability to the correct address space. In this case, the start/stop exit routine could do the following:

- Set up a register pointing to the CTSS.
- Issue a PC to the PC routine responsible for processing trace requests.
- Upon return, send the PC routine's return code back to component trace.

The PC routine responsible for processing trace requests could do the following:

- Use access register ASC mode to access the CTSS in the home address space.
- Parse any application-specific options if not already done in the start/stop exit routine.
- Process the request to start, stop, or change the trace options.
- Issue a program transfer (PT) or program return (PR) back to the start/stop exit routine with a return code.

See *z/OS MVS Programming: Extended Addressability Guide* for further information about PC routines.

Scheduling an SRB: The application can schedule an SRB to get addressability to the correct address space. In this case, the start/stop exit routine could do the following:

- Obtain extended SQA storage for:
  - An SRB
  - A copy of the CTSS and related control areas
  - An ECB.

- Initialize and schedule the SRB (SCHEDULE macro) to the application's address space.
- Wait for the ECB to indicate that the SRB has completed.
- Upon being posted and receiving a return code, send the return code back to component trace.

The SRB routine could do the following:

- Parse the application-unique options.
- Process the request to start, stop, or change the trace options.
- Post the start/stop routine and provide a return code.

See Chapter 9, "Using a service request block (SRB)," on page 195 for further information.

Entry specifications

Component trace passes information to the start/stop exit routine in a parameter list and in GPRs.

Registers at entry: On entry to the start/stop routine, the GPRs and access registers (ARs) contain the following information:

Register	**Contents**
GPR 0	No applicable information
GPR 1	Address of the standard parameter list, which contains the address of the component trace parameter list (CTSS).
GPRs 2 - 12	If you use these registers, you must save and restore the contents.
GPR 13	Address of a 72-byte register save area, located in pageable, private-area storage.
GPR 14	Return address.
GPR 15	Entry point address of the start/stop routine.
ARs 0 - 15	If you use these registers, you must save and restore the contents.

Parameter list: The standard parameter list consists of one element: the address of the component trace parameter list (CTSS). Figure 17 on page 120 illustrates some of the tables the CTSS points to, including:

- A table containing the options from an operator REPLY or parmlib member (CTSSOPTP)
- A table containing the ASIDs from an operator REPLY or parmlib member (CTSSASIP)
- A table containing the JOBNAMEs from an operator REPLY or parmlib member (CTSSJOBP)
- A sublevel table (CTSSSNTP) if sublevels are involved in the request.

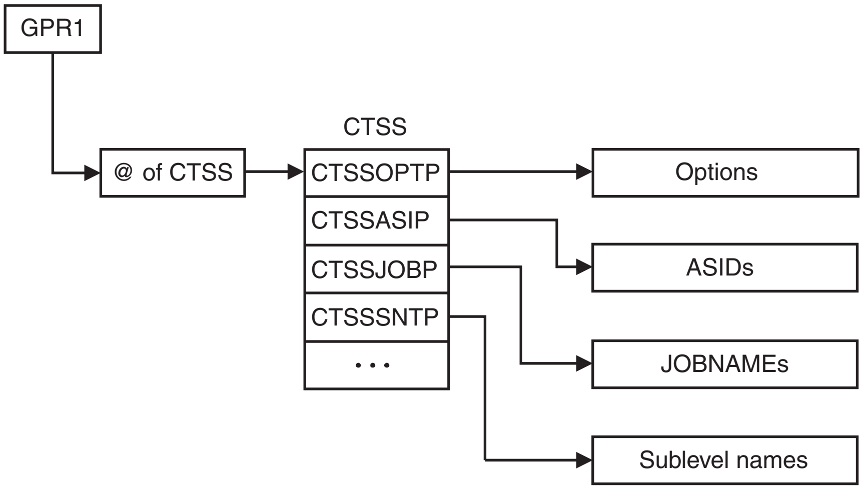


Figure 17. Information Passed to the Start/Stop Exit Routine Through GPR 1

The CTSSUSRD field of CTSS contains the 16 bytes of information specified on the USERDATA parameter of CTRACE DEFINE. This field will contain zeros if not specified on CTRACE DEFINE. This CTSS user data is treated as input to and output from the start/stop routine, and can be changed by the start/stop routine.

The standard parameter list, the CTSS, the OPTIONS buffer, the ASID table, and the JOBNAME table are located in pageable, private-area storage. The CTSS is mapped by the ITTCTSS mapping macro. For the field descriptions provided by the ITTCTSS mapping macro, see CTSS in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Return specifications

On return to component trace, the start/stop routine must place information in the registers. Component trace takes action based on the return codes set by the routine.

Registers on return: On return to component trace, the start/stop routine must place the following information in the registers:

Register	Contents
GPR 0	Reason code, created by the application program, if the return code is not zero.
GPR 1	No requirement.
GPRs 2 - 14	You must restore the contents of these registers.

Register	Contents
GPR 15	Return code. On return from a request to start tracing, IBM *recommends* you set the return codes as follows: **0** The start/stop routine completed successfully; the routine performed the specified action as requested on the TRACE CT operator command or on CTRACE DEFINE. **4** An option specified in the operator reply is in error. Register 0 contains a unique, user-defined, 4-byte reason code to identify the specific error. The operator can re-issue the command. **8** An environmental error. Register 0 contains a unique, user-defined, 4-byte reason code to identify the specific error. On return from a request to stop tracing, IBM recommends you set the return codes as follows: **0** Tracing was stopped or set to the minimum level. **Other** An error occurred.
ARs 0 - 15	You must restore the contents of these registers.

Component trace processing after return: If the start/stop routine sets a non-zero return code, the system displays both the return and reason codes through an operator message. For this reason, you should make your return and reason codes as meaningful as possible. The return codes shown in the above table are recommended by IBM.

When issuing CTRACE DEFINE, the application can include the SSRC and SSRSNC parameters to obtain the start/stop routine's return and reason codes.

Coding a display trace exit routine

An application that will be using component trace might have a display trace exit routine that provides status information about the component trace for the DISPLAY TRACE operator command. You are responsible for writing this optional routine.

A display trace exit routine is desirable if the component trace changes the trace mode, buffer size, or options independent of the TRACE CT operator command. There may be conditions under which the component trace may need to change the status of the trace, such as:

- If the operator issues the TRACE CT command to change the buffer size, the component trace might round the buffer size up or down. When a D TRACE command is displayed and a display trace exit routine is provided, the display exit can provide the buffer size to be displayed. If no display trace exit routine is provided, then the buffer size specified on the TRACE CT command is displayed.
- If a problem occurs, the component trace might decide to stop or start tracing, or change trace options.

Exit routine environment

The display trace exit routine receives control in the following environment:

Environmental factor	**Requirement**
Authorization:	Supervisor state and PSW key 0
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN = SASN
AMODE:	31-bit. All passed areas can be above 16 megabytes in virtual storage.
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Serialization:	Component trace prevents multiple conflicting commands from being processed simultaneously.
Location:	The display routine must be in LPA or the LNKLST concatenation. If the display routine is in LPA, the system passes control to the module at that location, and the module is not page-fixed. Otherwise, the system places the display routine into the fixed common service area (CSA).

Exit recovery: The display routine must supply its own ESTAE-type recovery routine or functional recovery routine (FRR) for any needed recovery or cleanup of resources.

Exit routine processing

The display routine receives control to provide status information about the component trace for the DISPLAY TRACE operator command. The CTSSDSPI bit indicates that trace display information is to be provided.

When the display routine receives control it should:

- Set one of the following bits in the CTSS to indicate trace mode:
  - CTSSDON - to indicate that the trace mode is ON.
  - CTSSDOFF - to indicate that the trace mode is OFF.
  - CTSSDMIN - to indicate that the trace mode is MIN.
- Parse the options in the OPTIONS parameter passed to the routine. These are the options that were specified by the operator on a previously issued TRACE CT command either by operator reply or in a CTncccxx parmlib member. If the options being traced are different than those passed to the routine, the routine should update the CTSS with the trace options in effect.
- If the routine detects errors, it can issue a WTO using the console ID and CART specified in field CTSSBUFS, and set a failing return code.
- Set the trace buffer size in the CTSS.
- Use the 16 bytes of user data in the CTSSUSRD field. The application can specify this user data on CTRACE DEFINE with the USERDATA parameter, or in the CTSSUSRD field when the start/stop routine gets control.
- If the routine detects no errors, set the return code to zero.
- Establish the returned general purpose registers. See "Return specifications" on page 120 for the return specifications.
- Return control to component trace.

Programming considerations

When writing the display trace exit routine, component trace displays the mode, buffer size, and options that were previously set (either from a previous TRACE CT command or from initial IPL), if either:

- the display trace exit routine does not set either the CTSSDON, CTSSDOFF, or CTSSDMIN bits
- or the display trace exit routine returns a non-zero return code.

Exit routine communications

The display trace exit routine runs in a system address space.

The routine might have to communicate with another address space to obtain the display information to be returned to component trace. See "Start/stop exit routine communications" on page 117 for methods to accomplish this communication.

Entry specifications

Component trace passes information to the display exit routine in a parameter list and in General Purpose Registers (GPRs).

Registers at entry: On entry to the display routine, the GPRs and access registers (ARs) contain the following information:

Register	Contents
GPR 0	No applicable information
GPR 1	Address of a standard parameter list, which contains the address of the component trace parameter list (CTSS).
GPRs 2 - 12	If you use these registers, you must save and restore the contents.
GPR 13	Address of a 72-byte register save area, located in pageable, private-area storage.
GPR 14	Return address.
GPR 15	Entry point address of the display routine.
ARs 0 - 15	If you use these registers, you must save and restore the contents.

Parameter list: The standard parameter list consists of one element: the address of the component trace parameter list (CTSS). Figure 18 on page 124 illustrates some of the tables the CTSS points to, including:

- A table containing the options from an operator REPLY or parmlib member (CTSSOPTP)
- A sublevel table (CTSSSNTP) if sublevels are involved in the request.

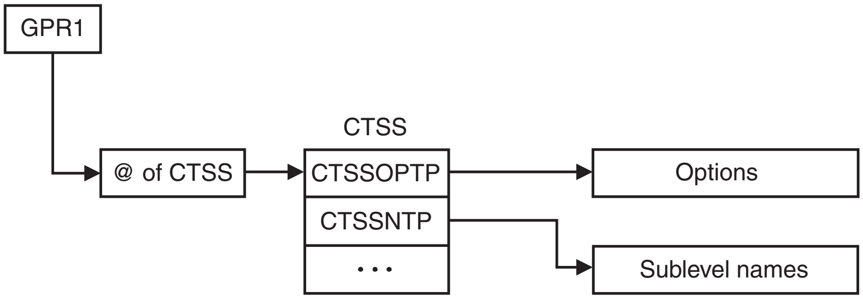


Figure 18. Information Passed to the Display Trace Exit Routine Through GPR 1

The CTSSUSRD field of CTSS contains the 16 bytes of information specified on the USERDATA parameter of CTRACE DEFINE. This field will contain zeros if not specified on CTRACE DEFINE. This CTSS user data is treated as input to and output from the display routine, and it can be changed by the display routine.

The standard parameter list, the CTSS, and OPTIONS buffer table are located in pageable, private-area storage. The CTSS is mapped by the ITTCTSS mapping macro. For the field descriptions provided by the ITTCTSS mapping macro, see CTSS in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Return code specifications

On return to component trace, the display routine must place information in the registers. Component trace takes action based on the return codes set by the routine.

Registers on return: On return to component trace, the display routine must place the following information in the registers:

Register	Contents
GPR 0	No requirement.
GPR 1	No requirement.
GPRs 2 - 14	You must restore the contents of these registers.
GPR 15	Return code. On return from a request to provide information, IBM *recommends* you set the return codes as follows: **0** The display routine completed successfully; the routine updated the CTSS with the trace display information. **Other** An error occurred.
ARs 0 - 15	You must restore the contents of these registers.

Component trace processing after return: If the display routine sets a non-zero return code, or it does not set either the CTSSDON, CTSSDOFF or CTSSDMIN bits, the system displays the previous trace information and the trace information returned by the display trace exit routine in CTSS is ignored.

Creating trace entries

This information describes:

- Creating component trace entries (CTEs) and placing them in trace buffers
- Understanding the fields in a CTE
- Organizing the variable data in a CTE.

Your trace entries can be in any format. However, to be able to use IPCS to display them, use the ITTCTE mapping macro to create CTEs. You can trace a maximum of 60KB of application-unique data in each CTE.

Figure 19 is an example of a CTE with a trailing length, which is a length field at the end of the entry.

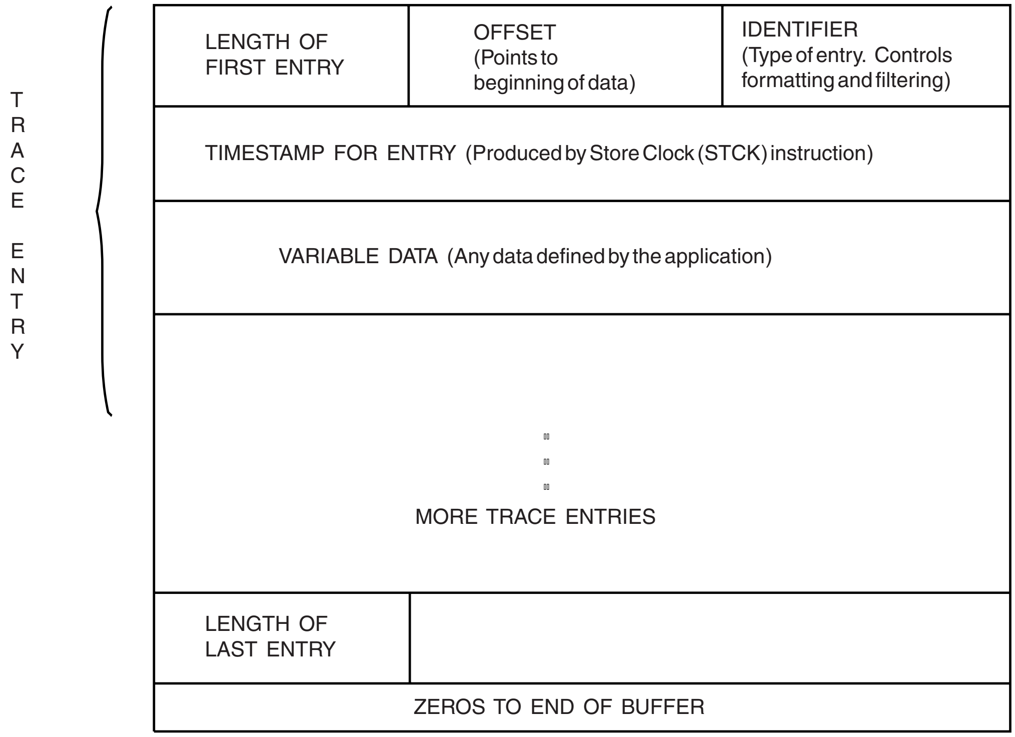

TRACE ENTRY
LENGTH OF FIRST ENTRY
OFFSET (Points to beginning of data)
IDENTIFIER (Type of entry. Controls formatting and filtering)
TIMESTAMP FOR ENTRY (Produced by Store Clock (STCK) instruction)
VARIABLE DATA (Any data defined by the application)
MORE TRACE ENTRIES
LENGTH OF LAST ENTRY
ZEROS TO END OF BUFFER

Figure 19. Component Trace Entry in a Trace Buffer

Follow this procedure when creating your CTEs and placing them in buffers:

- Start the trace buffer with a complete CTE at offset 0.
- Continue adding CTEs without any gaps between the trailing length field (CTELENE) of the previous CTE and the starting length field of the next CTE (CTELENP).
- When the buffer reaches a stage where the entire next CTE cannot fit, IBM recommends that you clear the remaining portion of the trace buffer to zeros. At this point, you have the choice of either using a new buffer or wrapping and reusing the same buffer. If you are using the external writer, use a new buffer so that component trace can capture your full trace buffer asynchronously while you are writing trace entries to a new buffer. If you do not require trace data to be captured asynchronously, or if you can afford to lose old trace data, you might reuse the same buffer.

Once the application creates all its trace entries, places them in trace buffers in storage, and externalizes them either in a dump or by using the external writer, IPCS can format each entry according to instructions you specify in a CTRACE format table. Use the FMTTAB parameter on CTRACE DEFINE to specify the name of the load module that contains the CTRACE format table. See *z/OS MVS IPCS Customization* for information about creating a CTRACE format table.

Understanding the fields in a CTE

The following fields are common in every CTE:

`Length (CTELENP)`
A 2-byte field containing the length of the entire CTE, including this length field (CTELENP) and the trailing length field (CTELENE). The maximum CTE size is 64KB. This field is at the beginning of the CTE.

`Offset (CTEOFF)`
A 2-byte field containing the offset of the variable data within the CTE.

`Entry Identifier (CTEFMTID)`
A 4-byte field containing a user-defined identifier. This indicates the type of event traced by the entry, and is also specified on the ITTFMTB macro EVENTDATA keyword in the corresponding format table entry for formatting purposes.

`Time (CTETIME)`
An 8-byte field containing the time stamp. IBM recommends that you create the time stamp using the STCK instruction, which produces Coordinated universal time (UTC). During formatting, the user of IPCS can request that the time stamps be formatted in local time only if you created the time stamps in UTC.

`Variable Data (CTEVDATA)`
A variable-length field to contain application-unique data in any format. The maximum length of this field is 60KB. See “Organizing variable data in CTEs” for suggestions on how to organize this field.

`Trailing Length (CTELENE)`
A 2-byte field containing the length of the entire CTE, consistent with the value in CTELENP. This field is at the end of the CTE.

For complete field names and lengths, offsets, and descriptions of the fields mapped by the ITTCTE mapping macro, see ITTCTE in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Organizing variable data in CTEs

You must determine how you will organize the variable portion (CTEVDATA) of the information in the CTEs. This information discusses the following methods:

- Using a fixed-field format
- Making each entry unique
- Using a key-length data format.

You might combine these methods to produce other methods.

Using a fixed-field format: If you need the same type of data in every CTE, you can have one fixed definition of a trace entry. For example:

```
Offset 0  - ASID
Offset 2  - Module name
Offset 10 - Flags
Offset 14 - Return code
```

You might want to start every CTE with a few fixed fields, and follow that with variable data. Using fixed-field formats for CTEs allows for a smaller format table, and fewer lines of code, because you will not need separate format table entries to support each trace entry.

Making each entry unique: In this case, each CTE contains data that is unique, which requires unique formatting support and requires a larger format table than fixed-field formats. The advantage is that you can capture whatever data you require for a particular CTE.

Using key-length data format: With key-length data format, you define a key that identifies the data type, and the length indicates where the next key resides. This method is useful when your application supports a number of data types, but only a select number of these data types will be in each CTE.

Here is an example of a key-length data format:

```
KEY = 1 means ASID
LENGTH = 2 allows 2 bytes for an ASID in hexadecimal
DATA = 2 hexadecimal bytes for an ASID
```

Based on the above example, the formatting program will format an ASID when the key is 1. This method provides flexible and low-cost formatting (there is less code to format). One disadvantage is that you consume extra storage in the CTE for the key and length fields, and might consume extra instructions in the path that creates the CTEs.

Chapter 7. Communication

The following communication topics are included in this information:

- Interprocessor communication
- Writing operator messages
- Issuing operator commands
- Routing operator commands in a sysplex
- Extended MCS consoles.

Interprocessor communication

Interprocessor communication (IPC) is a function that provides communication between processors sharing the same control program. Those executing functions that require a processor or program action on one or more processors use the IPC interface to invoke the desired action. The IPC function uses the signal processor (SIGP) instruction to provide the necessary hardware interface between the processors.

Based on the condition code of the SIGP instruction, the IPC function may invoke the excessive spin routine. The excessive spin routine may cause message IEE331A to be issued. This message either requires the operator to initiate alternate CPU recovery (ACR) or continue with processing. For more information concerning the SIGP instruction, see *Principles of Operation*.

Remote class services are defined for those control program functions that require the execution of a software function on one of the configured processors. Emergency signal is a remote immediate service that can be invoked via the RISGNL macro.

An emergency-signal external-interruption condition is generated at the specified processor. The interruption condition becomes pending during the execution of the SIGP instruction. The associated interruption occurs when the processor is interruptible for that condition. At any one time the receiving processor can keep pending one emergency-signal condition for each processor of the multiprocessing system, including the receiving processor itself. Issue the RISGNL macro to invoke the emergency signal function.

Writing and Deleting Messages (WTO, WTOR, and DOM Macros)

The WTO and WTOR macros allow you to write a message to a display device, a printer, a hard-copy log, or a program that receives WTO and WTOR messages (such as an EMCS console or a subsystem console). Besides writing a message, WTOR allows you to request a reply from the operator who receives the message. The DOM macro allows you to delete a message that is already written to the operator.

Routing the Message

You can route a WTO or WTOR message to a console by specifying one or more of the following parameters:

- ROUTCDE to route messages by routing code

- CONSID to route messages by console ID
- CONSNAME to route messages by console name
- MSGTYP and MCSFLAG to route messages by message type.

The ROUTCDE parameter allows you to specify the routing code or codes for a WTO and WTOR message. The routing codes determine which console or consoles receive the message. Each code represents a predetermined subset of the consoles that are attached to the system, and that are capable of displaying the message. The installation defines the consoles that belong to each routing code. WTO and WTOR allow routing codes from 1 to 128. Routing codes 30 through 41 are reserved, and are ignored if specified. Routing codes 42 through 128 are available to authorized programs only, although the ROUTCDE parameter itself is available to non-authorized as well as authorized users. See any volume of *MVS System Messages* for more information on routing codes.

You can also use either the CONSID or CONSNAME parameter to route messages. These mutually exclusive parameters let you specify the ID or the name of the console that is to receive the message. When you issue a WTO or WTOR macro that uses either the CONSID or CONSNAME parameter with the ROUTCDE parameter, the message or messages will go to all of the consoles specified by both parameters.

You can use the SYSNAME, JOBID, or JOBNAME parameter to specify the system or job from which a WTO or WTOR message is issued. SYSNAME is mutually exclusive with the CONNECT parameter.

Only programmers familiar with multiple console support (MCS) should use the MSGTYP parameter, which is associated with message routing. The MSGTYP parameter is typically used for messages related to the MONITOR command. Specifically, MSGTYP (JOBNAMES) will queue message to consoles that have activated MONITOR JOBNAMES; MSGTYP (SESS) will queue messages to consoles that have activated MONITOR SESS; and MSGTYP (STATUS) will queue messages to consoles that have activated MONITOR STATUS.

The MCSFLAG parameter is used to specify various attributes of the message, such as whether the message is:

- For a particular console
- For all active consoles
- For the hardcopy log
- A command response

> **Note:** IBM recommends using desciptor code 5, rather than MCSFLAG, to indicate a command response. This is because MPF exits can change the descriptor code, but can not change the MCSFLAG.

For the convenience of the operator, messages can be associated with individual keynames. A keyname consists of 1 to 8 alphanumeric characters, and it appears with the message on the console. The keyname can be used as an operand in the MVS DISPLAY R console command, which operators can issue at the console. Use the KEY parameter on the WTO or WTOR macro for this purpose.

Altering Message Text

The TEXT parameter on the WTO macro enables you to alter repeatedly the same message or numerous messages. You can alter the message or messages in one of two ways:

- If you issued 3 different messages, all with identical parameters other than TEXT, you created a list form of the macro, moved the text into the list form, then execute the macro. Using the TEXT parameter you can use the standard form of the macro, and specify the address of the message text. By reducing the number of list and execute forms of the WTO macro in your code, you reduce the storage requirements for your program.
- If you need to modify a parameter in message text, using the TEXT parameter enables you to modify the parameter in the storage that you define in your program to contain the message text, rather than modify the WTO parameter list.

Using the TEXT parameter on WTO can reduce your program's storage requirements because of fewer lines of code or fewer list forms of the WTO macro.

Writing a Multiple-Line Message

To write a multiple-line message to one or more operator consoles, either issue WTO with all lines of text, or issue each line of text separately using the CONNECT parameter on the WTO macro.

The CONNECT parameter connects a subsequent message to a previous message. For example, if your program develops a large, multiple-line message of unknown length, it can issue several WTOs for the different parts of the message at different times. The CONNECT parameter forces all these WTOs to use the same message ID, and physically unites the different parts of the message at the display console as a single message. CONNECT is mutually exclusive with CONSID, CONSNAME, and SYSNAME, and SYNCH=YES, and it is not available with WTOR.

You can create with one WTO macro request a message that consists of up to 255 lines. For more than 255 lines, issue more than one WTO macro. The additional lines appear at the end of the message and continue until you specify an “END” line by specifying “DE” or “E” as the line type for the last line of data. For the first request, you must ensure that register 0 is set to zero. If register 0 is not set to zero, WTO assumes that the multiple-line request is adding lines to an existing message, and no new message is created.

After processing the first request, the system places a message identifier in register 1. For each additional request, you must pass this identifier to the subsequent lines through the CONNECT parameter of WTO.

Embedding Label Lines in a Multiple-Line Message

Label lines provide column headings in tabular displays. You can change the column headings used to describe different sections of a tabular display by embedding label lines in the existing multiple-line WTO message for a tabular display.

Authorized programs (supervisor state, or PSW key 0-7, or APF-authorized) that are authorized to add lines to an existing multiple-line WTO message are also permitted to embed label lines within that existing multiple-line WTO message.

The label line does not have to appear immediately following the control line and before the data lines. At most two label lines can appear consecutively without an intervening data line.

Issuing a Message and Loading a Wait State (WTO and LOADWAIT Macros)

There are situations in which system operations cannot continue until the system operator takes some external action. An example might be an authorized application detecting a critical problem that warrants stopping the entire system to correct. Using the LOADWAIT macro and the WTO macro with the WSPARM parameter stops the system so the operator can correct a problem, if possible. By using LOADWAIT and WTO, you issue a message to the operator and place the system into a wait state. By placing the system into a wait state, all processor activity is stopped.

To place the system into a wait state:

- Issue the LOADWAIT macro to store wait state information into a parameter list.
- Issue the WTO macro and specify the parameter WSPARM to issue a message and load the wait state.

The WSPARM parameter contains the address of the parameter list that you previously built using the LOADWAIT macro. WTO issues one message to the operator and uses the parameter list from LOADWAIT to put the system into the wait state. The wait state code and operator message explain what action the operator is to take. For more information about wait state codes, see *z/OS MVS System Codes*.

Using the Wait State Macro (LOADWAIT)

Through the LOADWAIT macro, you specify the following wait state information:

- Wait state type - This must be either restartable or non-restartable.
- Wait state code - This is the wait state code you must assign for the error.
- Reason code - This is the reason code you can assign for further information about the wait state.
- Action code location - This is a 1-byte area to receive information that the operator can supply to the calling program.
- Additional information to be made available to the operator - This could be a system ID or a pointer to a data area for further information about the wait state. For more information about action codes, see the specific wait state codes in *z/OS MVS System Codes*.

Non-restartable and Restartable Wait States

You must specify on the LOADWAIT macro whether the wait state is non-restartable or restartable:

- A **non-restartable** wait state allows the calling program to stop the system. This type of wait state is used only when MVS cannot be allowed to continue operating. To continue processing, the operator must re-IPL the system.
- A **restartable** wait state allows the calling program to communicate with the operator (PSAPARM), allows the operator to communicate with the calling program (ACTCODE), and prevents any other activity in the system during operator communications. This type of wait state is used when:
  - MVS must be stopped until the operator corrects some external condition, but can continue after the condition is corrected.

  - MVS should be stopped to preserve storage contents for problem determination, but can continue afterward. An example of this type of wait state is when a SLIP trap (ACTION=WAIT) has been matched.

To continue processing, the operator initiates a restart action from the system console.

Invoking the LOADWAIT Macro

How you use the LOADWAIT macro depends on the information you have:

- If you have all wait state information at program assembly time, invoke LOADWAIT once to define and initialize the parameter list.
- If any wait state information is missing at program assembly time, invoke LOADWAIT separately, to:
  - Set up storage for the parameter list.
  - Add the wait state information. This step might require more than one invocation of the macro.

Example with All Wait State Information Known at Assembly Time

If you know the wait state and the reason code at program assembly time, the parameter list storage can be defined and initialized with **one invocation** of the LOADWAIT macro. The following example defines storage for and initializes a parameter list, and names the parameter list ‘TEST’:

```
LOADWAIT  MF=(L,TEST),
      CODE=WAITA29,
      WAITTYPE=RESTARTABLE
      REASON=SOMERC
```

Figure 20. LOADWAIT Example for Defining and Initializing a Parameter List

Issue the WTO macro with the WSPARM option to load a restartable wait state code of X'A29':

```
      WTO  'WMM0001  UNABLE TO COMMUNICATE A REPLY',
             LINKAGE=BRANCH,SYNCH=YES,WSPARM=TEST

WAITA29  EQU X'A29'
SOMERC   EQU X'3'
```

Figure 21. WTO Example for Issuing a Message with a WSPARM parameter

This example represents the most common usage of the LOADWAIT macro. The list form of the macro is used because both the wait state and reason code are known at program assembly time. The calling program specifies the LOADWAIT parameter list (TEST) on the WSPARM parameter of the WTO. WTO issues the message "WMM0001 UNABLE TO COMMUNICATE A REPLY" and loads the wait state.

Example with Wait State Information Not Known at Assembly Time

If you do not know all the wait state information at program assembly time, at least **two separate invocations** of the macro are required. In the following example, assume that the wait state code is passed to the calling program at execution time. Define storage for a parameter list named ‘SAMP’:

```
LOADWAIT MF=(L,SAMP)
```

Figure 22. LOADWAIT Example for Defining Storage for a Parameter List

The example uses the list form of the macro to reserve storage for the parameter list. You do not initializing the parameter list because you do not know the wait state code. Once you know the wait state, invoke LOADWAIT again to insert it into the parameter list as follows:

```
LOADWAIT MF=(M,SAMP),
   CODE=WAIT,
   WAITTYPE=RESTARTABLE
```

Figure 23. LOADWAIT Example for Modifying an Existing Parameter List

Notice that Figure 23 uses the modify form of the LOADWAIT macro because you are storing the wait state code into the existing parameter list, which you reserved with Figure 22.

Issue the WTO macro with the WSPARM option to issue a message and load a restartable wait state with the wait state code that was passed from the calling program:

```
      WTO  'SCM0001  WTOR FAILED',
            LINKAGE=BRANCH,SYNCH=YES,WSPARM=SAMP

WAIT  DS H
```

Figure 24. WTO Example for Issuing a Message and Loading a Wait State

When you use the modify form of the macro, the parameter list is reset to all zeros every time. **If you use the modify form of the LOADWAIT macro, be sure to specify all the required information**.

For example, if you wanted to change the wait state type from restartable to non-restartable and add a reason code to the parameter list named ‘SAMP’ after your previous modify invocation, you must respecify the required fields that you've already coded:

```
LOADWAIT MF=(M,SAMP),
   CODE=WAIT,
   REASON=BADRC,
   WAITTYPE=NONREST
```

Figure 25. LOADWAIT Example for Adding a Reason Code to a Parameter List and Changing the Type of Wait State

Issue the WTO macro with the WSPARM option to issue a message and load a non-restartable wait state:

```
      WTO  'WMM0009  BAD RETURN CODE ENCOUNTERED',
            LINKAGE=BRANCH,SYNCH=YES,WSPARM=SAMP

WAIT   DS H
BADRC  DC X'07'
```

Figure 26. WTO Example of a Message with a WSPARM Parameter

Notice that Figure 25 uses the modify form of the LOADWAIT macro because you are respecifying the wait state information into the existing parameter list you reserved with Figure 22. This time (Figure 25) the example adds a reason code and makes the wait state non-restartable. Because the parameter list is reset to zeros when using the modify form of the LOADWAIT macro, you must respecify all the previous information about the wait state that you had previously coded. Because this example shows a non-restartable wait state, the operator must re-IPL the system to continue processing.

Deleting Messages Already Written

The DOM macro deletes the messages that were created using the WTO or WTOR macros. Depending on the timing of a DOM macro relative to the WTO or WTOR, the message may or may not have already appeared on the operator's console.

- When a message already exists on the operator screen, it has a format that indicates to the operator whether the message still requires that some action be taken. When the operator responds to a message, the message format changes to remind the operator that a response was already given. When DOM deletes a message, it does not actually erase the message. It only changes the message‘s format, displaying it like a non-action message.
- If the message is not yet on the screen, DOM deletes the message before it appears. The DOM processing does not affect the logging action. That is, if the message is supposed to be logged, it will be, regardless of when or if a DOM is issued. The message is logged in the format of a message that is waiting for operator action.

The program that generates an action message is responsible for deleting that message.

To delete a message, identify the message by using the MSG, MSGLIST, or TOKEN parameters on the DOM macro, and issue DOM.

When you issued WTO or WTOR to write the message, the system returned a message ID in general purpose register 1. Use the ID as input on the MSG or MSGLIST parameters. MSGLIST (message list) associates several message IDs with the delete request. You define the number of message IDs in the message list by the COUNT parameter or by specifying a 1 in the high-order bit position of the last message ID in the list. The count parameter cannot exceed 60. If you specified the TOKEN parameter on one or more WTOs or WTORs, use the same value on the TOKEN parameter of DOM to delete all of those messages.

Writing to the System Log

There are three ways to request that the system write a message to the system log:

- Your installation may have specified, through SYS1.PARMLIB, that all messages appear at the device that is the hard-copy log.
- Use the WTL (Write-to-Log) macro.
- Use the HRDCPY option on the MCSFLAG parameter on the WTO macro.

IBM recommends that you use the WTO macro to write to the system log.

Issuing an internal START or REPLY command (MGCR)

To issue an internal START or REPLY command, a program can issue the MGCR macro. The program issuing MGCR can pass a user security token to the system. The system uses the user security token for command authorization checking. In the case of the START command, the program that issues MGCR can pass 31 bits of information (in a field called a program token) to the program being started. An internal REPLY command is available to reply to a WTOR message.

You can also issue an internal START or REPLY command by using the MGCRE macro. MGCRE allows you to issue any operator command from an MCS or extended MCS console. MGCRE is the preferred programming interface. See "Issuing operator commands from a program (MGCRE macro)" on page 137 for more information on using the MGCRE macro.

Before issuing the MGCR macro, set general register 0 to zero, and initialize a buffer as follows:

1 byte	1 byte	2 bytes	variable length	4 bytes	80 bytes
flags1	length	flags2	text	Program token right justified	user security token

flags1
If bit 0 of the flags1 byte is one, the flags2 field must contain meaningful information. Bits 1-7 of flags1 must be zero.

length
The length field contains the length of the buffer in bytes, up to but not including the program token field.

flags2
Use this field to indicate whether a program token, a user security token, or both are present:

- X'0000' - neither a program token nor a user security token is present.
- X'0800' - a program token is present.
- X'0008' - a user security token is present.
- X'0808' - both a program token and a user security token are present.

Note: Bit 0 of **flags2** must be zero.

text
The text field contains the START or REPLY command followed by operands and, optionally, comments. The field may be up to 126 bytes long.

program token
This field contains information that you want to pass to the started program. You can pass any information depending upon the design of your program. For example, you might pass a parameter or a pointer to a parameter list. Program token is meaningful only for the START command.

user security token
Use this field to specify the user security token the system is to use for command authorization checking. To obtain the user security token, use the REQUEST=TOKENXTR form of the RACROUTE macro. For a description of the RACROUTE macro, see *z/OS Security Server RACF Macros and Interfaces*.

Figure 27 on page 137 shows an example of how to setup the buffer. For a description of the MGCRPL data area (mapped by IEZMGCR), see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

```
           SR   REG0,REG0        INDICATE SYSTEM ISSUED
  *                                COMMAND
MGCRMAC    MGCR MGRCDATA
           .
           .
MGCRDATA   EQU  *
FLG1       DC   X'80'
LGTH       DC   AL1(PTOKEN-MGCRDATA)
FLG2       DC   X'0808'
TXT        DC   C'S IMS ***ANY COMMENTS***'
PTOKEN     DC   AL4(ECB)         ECB ADDR
UTOKEN     DC   CL80 '***YOUR SECURITY TOKEN***'
```

Figure 27. Setting Up the Buffer for MGCR

Issuing operator commands from a program (MGCRE macro)

The MGCRE macro enables you to issue commands and responses to messages without operator intervention. For example, an application can issue a VARY or CONTROL command by using MGCRE, which might satisfy an outstanding action message.

Using MGCRE to issue internal START and REPLY commands provides many benefits over the MGCR macro. If you use MGCR, you must initialize a parameter list before issuing the macro. You do not have to do this initialization if you use MGCRE. MGCRE builds a parameter list for you, based on macro parameters. MGCR is limited to only START and REPLY commands, whereas MGCRE will accept any operator command.

NOT Programming Interface information

The CMDFLAG=TSO option allows TSO to use MGCRE instead of MGCR. CMDFLAG=TSO causes the CONSID value in the parameter list to be treated as a TSO identifier, instead of a console ID.

End of NOT Programming Interface information

Issuing a command response message

Many IBM, customer, and vendor programs process operator commands. These command processors typically issue one or more WTO macros "in response to" the command.

When programs issue commands, they use the MGCRE macro (see “Issuing operator commands from a program (MGCRE macro)”), employing logic similar to the following:

1. Issue a command with "my" console ID, and a command and response token (CART) value uniquely related to this command.
2. Wait to receive a response message (marked as a response), back to "my" console, with the same CART value as the command input.
3. Normally, consider the command "finished" when the first response is received. If a command processor issues more messages, the logic is more difficult.

Rules for a command response WTO

Here are some general rules to follow when coding a WTO in response to a command.

1. Specify descriptor code 5 (DESC=5)
   - Identifies message as a command response
   - Can have additional descriptor codes, if wanted

   "Extended" (program) consoles have an option to receive only command responses, and ignore "unsolicited" messages. These programs depend on a command response message being marked as such.

   "Unsolicited" messages are those which are directed to a console because of routing codes, or some other routing attribute. They have not been directly "solicited" by the operator. Command responses are considered "solicited" messages.
2. Direct the message to the console which issued the command (CONSID=*cccc*), using the 4-byte console ID of the console which issued the command. All consoles have a 4-byte console ID, which can be used to direct the message to the issuing console.
3. Return the CART value specified on the command (CART=*vvvvvvvv*)

   CART data is an 8-byte field passed to you from the input command.

   All programs which use the MGCRE macro to issue commands can specify a Command And Response Token (CART) to identify "which command" this is a response to.

   If the command issuer did not use a CART, it is still safe to say "CART=" on the WTO, and point to the CART field in your control block. In this case, the CART is propagated as a field of zeroes.
4. Issue one and only one message in response to the command.

   It is preferable to respond with one message, which may be many lines of text. You can issue a multiple line message "all at once", or use the CONNECT parameter. Don't forget to "end" the message (issue an end line or data-end line).

   Be sure to clear register zero before issuing a WTO for a line (or block of lines) of a multiple-line message. If MVS finds a non-zero value in register zero, it will assume you are trying to CONNECT to a message with that message ID.

Old code conversion

If you are converting existing messages, there are a few things you may see:

- MCSFLAG=RESP

  This keyword might be used instead of DESC=5 to indicate a command response. This is supported, but DESC=5 is preferred, and the WTO should be changed if convenient.

  Descriptor codes can be changed by the installation's message exits, but MCSFLAGs cannot. For maximum flexibility for the customer, DESC=5 is preferred.
- MCSFLAG= REG0 or QREG0

  Old code might have the "register zero interface" (REG0 or QREG0). If these keywords are specified, the console ID is placed in the low-order byte of register zero before the WTO is issued.

  Only a 1-byte console ID can be specified this way, so this code must be converted to use CONSID=.

Note:

1. As of V1R8, REG0 and QREG0 are no longer supported because of the removal of 1-byte console IDs.
2. The high-order 3 bytes of register zero are assumed to be a CONNECT ID, if this is a multiline message.

(Note:).

- Multiple messages

  You may be issuing several messages (instead of one multiline) in response to a command. This should not be changed. Changing the messages in this way would be a serious incompatibility for any automation programs which were analyzing the messages.

  "New" messages should be designed as a single multiline response, if possible.

Where to get the information

There are several ways for a command processing program to receive its input command; you can find your console and CART information in the same control block as the command text.

CIB control block (mapped by IEZCIB)

- CIBX (CIB extension) is found by adding CIBXOFF offset value to the address of the CIB.
- 4 byte console ID is CIBXCNID
- CART is CIBXCART

CSCB control block (mapped by IEECHAIN)

- 4 byte console ID is CHCNSIDI.
- CART is CHCART.

CMDX control block (mapped by IEZVX101)

This is the input to the command exit routines.

- 4 byte console ID is CMDXC4ID.
- CART is CMDXCART.

SSCM control block (mapped by IEFSSCM)

This is the input to the command subsystem interface routines.

- 4 byte console ID is SSCMCNID.
- CART is SSCMCART.

Assembler example with CIB control block

This is a simple example of a WTO, where the text is not variable. It is also not re-entrant code.

```
Assume the address of the CIB has been previously loaded into R7
    .
    .
    .
    USING   CIB,R7            CIB based on R7
    LR      R8,R7             Set up R8 as address of CIBX
    AH      R8,CIBXOFF        CIBX=addr(CIB)+CIBXOFF
    USING   CIBX,R8           Addressability to CIBX
    WTO     'ABC123I XYZ COMMAND COMPLETED SUCCESSFULLY',          X
            DESC=5,                                                X
            CONSID=CIBXCNID,                                       X
            CART=CIBXCART
    DROP    R8                done with CIBX
```

```
CIB    DSECT
       IEZCIB
       .
       .
+CIBXCART DS CL8 - COMMAND AND RESPONSE TOKEN
+CIBXCNID DS F   - CONSOLE ID
       .
       .
```

Assembler example with CMDX control block, multi-line WTO

This is an example of a multiple-line WTO using CONNECT. The text is variable. This code is re-entrant, if the static data has been copied into a getmained area.

```
       .
       .
* Access to CMDX parameter list
       L       R8,0(R1)           save address of CMDX
       USING   CMDX,R8            Access the CMDX
       .
       .
       .
* Put the first line of text into the TEXT field
* Set the length of the first line
         LA    R4,L'first line of text
         STH   R4,TEXTLEN
* Copy actual first line into text field
         MVC   TEXTTEXT(L'first line of text),first line of text
* Clear register 0 so it will not look like a CONNECT ID
         XR    R0,R0
* Issue WTO for the first line
         WTO   TEXT=((TEXTADDR,,)),                                    X
               CONSID=CMDXC4ID,                                        X
               CART=CMDXCART,                                          X
               DESC=5,                                                 X
               MF=(E,LINE1)
* Save the connect ID returned from the first line WTO
         ST    R1,MSGID
*
* Repeat the following block until finished putting out all the lines
*
* Put the next line of text into the TEXT field
* Set the length of the next line
         LA    R4,L'next line of text
         STH   R4,TEXTLEN
* Copy actual next line into text field
         MVC   TEXTTEXT(L'next line of text),next line of text
         WTO   TEXT=((TEXTADDR,,)),                                    X
               CONNECT=MSGID,                                          X
               MF=(E,CONTINUE)
*
* Ship a "null end line" (no text) to indicate message is finished
         WTO   ('',E),                                                 X
               CONNECT=MSGID,                                          X
               MF=(E,NULLEND)
               .
               .
               .
* Dynamic variables - declare as static and copy into getmained area
LINE1     WTO  TEXT=((,D)),                                            X
               CONSID=,                                                X
               CART=,                                                  X
               DESC=5,                                                 X
               MF=L
CONTINUE  WTO  TEXT=((,D)),                                            X
               CONNECT=,                                               X
               MF=L
```

```
NULLEND   WTO  ('',E),                                              X
               CONNECT=,                                            X
               MF=L
TEXTADDR DS 0H
TEXTLEN  DC CL2' '
TEXTTEXT DC CL72' '
MSGID    DS F
               .
               .
               .
* DSECTs
* CMDX - Command exit parameter list                              -*
        IEZVX101                  CMDX
```

Controlling command flooding (IEECMDS macro)

The IEECMDS macro provides the same functions as the command CMDS; either can be used to display and manipulate the commands which are attached or waiting to be attached. It can be used to remove commands, but it cannot be used to cancel commands which are already executing.

When the macro is executed with REQUEST=INFO, the response is returned in the user-specified BUFFER parameter. The IEEZB889 macro maps the information returned.

Routing commands in a sysplex (CPF macro)

The command prefix facility (CPF), whose entries are defined through the CPF macro, allows any operator or any authorized application to enter a command from any system in a sysplex and route that command to the appropriate system for execution.

Note: If the installation has defined the security profile MVS.CPF.ROUTE.CHECK in the OPERCMDS class, the issuer of the command requires sufficient authority to the MVS.ROUTE.CMD.system to route the command to a different system in the sysplex.

The command responses will come back to the originating console. The application can be an installation exit, a subsystem, or an installation-written program.

CPF assigns unique installation-defined prefixes to each system so that an application can direct a command to any system in the sysplex, just as if that system is the one on which the application is running. For example, if your installation is running in a poly-JES2 environment, you can use CPF to create a unique command prefix for each copy of JES2, so that you direct JES2 commands to a specific copy of JES2. When an application with CPF REQUEST=DEFINE in its code initializes, CPF adds to the CPF table the prefix for the system to which the application will direct commands. The CPF table is shared by all systems in the sysplex. Also, by issuing the CPF macro, an application can modify or delete entries in the table.

CPF also ensures that two or more systems do not have the same or overlapping prefixes, which helps prevent confusion. To ensure the integrity of your command prefixes, use CPF for as many prefix definitions as possible in your system.

Assigning a prefix

To define a valid prefix that will not conflict with existing prefixes, adhere to the following guidelines:

- Define a prefix as a 1- to 8-character string.
- Do not use a prefix that is a command, an abbreviation of a command, or a command invocation; for example:
  - “D” conflicts with MVS commands such as DISPLAY
  - “$TP1,Q=A” conflicts with a command invocation.
- Do not define a prefix that is either a subset or a superset of an existing prefix with the same first character. For example, if command prefix $XYZ already exists, command prefixes $, $X, and $XY are subsets of, and conflict with the original prefix. Similarly, prefixes $XYZ1 and XYZ will also conflict with existing prefix $XYZ because they are supersets with the same first character. You can, however, define command prefixes XYZ, YZ, or Z, because they do not start with the same first character as the existing prefix. You can see which prefixes already exist by issuing the DISPLAY OPDATA command.
- Define a prefix using characters from the set of valid characters listed in Table 19. This list is not exhaustive as the CPF macro will allow any hex value from X'41' to X'FE'.

Table 19. Valid Character Set

Character Set	Contents	
Alphanumeric	alphabetic numeric	Uppercase A through Z 0 through 9
National (See note)	“at” sign dollar sign pound sign	@, $, # (Characters that can be represented by hexadecimal values X'7C', X'5B', and X'7B')
Special	comma, period, slash, apostrophe, left parenthesis, right parenthesis, asterisk, ampersand, plus sign, hyphen, equal sign, cent sign, less than sign, vertical bar, exclamation point, semi-colon, percent sign, underscore, greater than sign, question mark, colon, quotation marks	, . / ' ( ) * & + - = ¢ < | ! ; % _ > ? : "

Note: The system recognizes the following hexadecimal representations of the U.S. National characters: @ as X'7C', $ as X'5B', and # as X'7B'. In countries other than the U.S., the U.S. National characters represented on terminal keyboards might generate a different hexadecimal representation and cause an error. For example, in some countries the $ character may generate a X'4A'.

Persistence of the prefix

The FAILDISP keyword on the CPF macro defines what the system will do on behalf of the prefix owner when the subsystem on which the command will be processed terminates:

- If you specify FAILDISP=RETAIN, the system will keep the prefix in the CPF table at all times, even if the subsystem owning the prefix terminates. In this case, you can decide what to do with the command prefix. You have three options:
  - Perform no action
  - Delete the prefix from the CPF table (REQUEST=DELETE)
  - Transfer ownership of the prefix to a system or subsystem in the sysplex using the REDEFINE keyword.

  If you do nothing, subsequent use of the command prefix will cause an error message. If you redefine the owner of the command prefix, you must update the CPF table to reflect the new owner.
- If you specify FAILDISP=PURGE, the system will delete the command prefix from the CPF table.
- If you specify FAILDISP=SYSPURGE, you will need to delete the command prefix from the table when either the subsystem owning the command prefix goes through end-of-memory (EOM) termination, or you remove from the sysplex the system to which CPF directs commands.

What is an extended MCS console?

An extended MCS console is a program that acts as a console. It can issue MVS commands, and receive command responses, unsolicited message traffic, and the hardcopy message set. There are two ways to use extended consoles:

- Interactively through IBM products such as TSO/E and Netview. See the documentation on those products for more information.
- Through an application program that you write. Examples of application program uses are:
  - Receiving automated message traffic
  - Defining a unique presentation service for messages to consoles

This information discusses how an application program can establish itself as an extended MCS console.

To establish a program as an extended MCS console, the program must issue the MCSOPER macro. Once activated as an extended MCS console, a program can receive messages and command responses by issuing the MCSOPMSG macro, and can issue commands by issuing the MGCRE macro. Unlike a standard MCS console, the extended MCS console can control which command responses it receives.

To receive messages and command responses, a program must issue the MCSOPMSG macro. Parameters on the MCSOPMSG macro enable a program to receive only specific types of messages. “Receiving messages and command responses, and issuing commands” on page 150 describes some of the parameters on the MCSOPMSG macro.

To receive the hardcopy message set, a program must issue the MCSOPER macro with REQUEST=ACTIVATE and the HARDCOPY attribute specified in the OPERPARM parameter list. “Receiving the hardcopy message set” on page 153 describes how to do this.

To issue commands, a program must issue the MGCRE macro. For more information on the MGCRE macro, see “Issuing operator commands from a program (MGCRE macro)” on page 137.

In a sysplex, an extended console application can receive messages from any system or systems in the sysplex, or can send commands to any system or systems in the sysplex. There is no system-imposed limit on the number of extended MCS consoles.

Extended MCS consoles will consume system resources, whether active or inactive. Once an extended MCS console is activated, the console definition remains for the life of the sysplex, even if it is later deactivated. IBM recommends care in the definition and use of extended MCS consoles, because an application that makes inefficient use of extended MCS consoles can cause system performance degradation. Specifically, an application should make use of existing extended MCS consoles, rather than defining new consoles. With z/OS V1R7, you can remove the definition of an unneeded EMCS console by using the sample program IEARELEC, which is shipped in SYS1.SAMPLIB. See *z/OS MVS Planning: Operations* for a description of this program and how to it.

If the application does not need to receive messages from all systems in the sysplex (or only some of the systems in the sysplex), the MSCOPE of the console can be changed to reduce the amount of message traffic that the console receives.

See “Example of managing an extended MCS console session” on page 155 for sample code for an extended MCS console. Sample code that shows how to use an extended MCS console resides in IEAEXMCS in SYS1.SAMPLIB.

This information explains how to:

- Activate a program as an extended MCS console
- Receive messages and command responses
- Receive the hardcopy message set
- Issue commands
- Deactivate an extended MCS console.

You can use a security product such as RACF to protect the use of extended MCS consoles. For more information on using RACF for this purpose, see *z/OS MVS Planning: Operations*.

This information refers to several data areas and some of their fields. The data areas and their mapping macros are:

Data Area	Mapping Macro
MDB	IEAVM105
MDB prefix	IEAVG132
MCSOP	IEZVG111
MCSCSA	IEAVG131

All of these data areas and their fields are documented in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Activating an extended MCS console

The MCSOPER macro with REQUEST=ACTIVATE defines and activates an extended MCS console to the system. When control returns to the calling program, the location specified on the CONSID parameter and the console status area in MCSCSA contain a 4-byte console ID that represents the activated extended console. You must specify the address of a message ECB on the MSGECB parameter of the MCSOPER macro. MSGECB identifies the ECB that the system is to post when a message is queued for the extended MCS console. Issue MCSOPMSG when an ECB is posted. MCSOPMSG returns a pointer to the queued message so that you can locate it. See “Specifying console attributes” for information on specifying console attributes.

Note: For system performance reasons, IBM recommends that the message ECB be in common storage.

When you activate an extended MCS console, the system creates a data space to store messages. There is one data space for every address space with an active extended MCS console. Therefore, if an address space has two active extended MCS consoles, both share the same message data space. Note that the system deletes this data space upon deactivation of all the extended MCS consoles in the address space.

When activating an extended MCS console, you need to specify its attributes and determine how you want to store messages.

Specifying console attributes

When activating the extended console, you need to specify its attributes, such as its command authority, routing codes, message data space size, and whether it is to receive the hardcopy message set. These attributes are known as operator parameters. You can specify these attributes, or operator parameters, in one of three ways:

1. In the OPERPARM segment of the user profile of a security product, such as RACF
2. In data area MCSOP, mapped by IEZVG111, when you specify the OPERPARM parameter on the MCSOPER macro
3. Through system defaults.

The system checks for the attributes in the order listed above. First it looks in the user profile of the console defined through your security product. The profile contains the address of the OPERPARM parameter list. Then it checks the MCSOP data area. If you did not specify the OPERPARM parameter on MCSOPER, the system applies default values for the console attributes, also defined in the MCSOP data area.

If you choose to specify attributes in MCSOP, set the appropriate MCSOP fields, and specify the OPERPARM parameter on the MCSOPER macro when you activate the extended console.

If you choose to specify attributes through your security product's user profile, see the security product's documentation for information on specifying attributes in the OPERPARM segment. *z/OS MVS Planning: Operations* contains information on using a RACF user profile to identify operator parameters.

Note: When the RACF OPERCMDS class is not active, the OPERPARM segment on the RACF user profile is ignored.

You can override the console attributes specified in the user profile of the security product by turning on bit MCSOVRDY in the MCSOP data area.

The following table summarizes the attributes. For information on the options for each field, see the MCSOP description in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Operator Parameter	MCSOP Field Name	OPERPARM Subkeyword (in OPERPARM segment of security product)	Default	Meaning
Authority	MCSOAUTH	AUTH	INFO	Command authority
Routing codes	MCSORCDT	ROUTCODE	NONE	Routing codes
Message level	MCSOMLVL	MLEVEL	ALL	Levels of messages
Message format	MCSOMFRM	MFORM	M	Message format for display
Message scope	MCSOMSFG	MSCOPE	*ALL	System message scope in a sysplex
Command scope	MCSOCSNM	CMDSYS	Current system	Command scope in the sysplex
Monitor information	MCSOMSGT	MONITOR	NONE	Receive status messages about given system events
Log command responses	MCSOLOGC	LOGCMDRESP	SYSTEM	Logging of command responses
Storage	MCSOSTOR	STORAGE	1	Limit of storage in megabytes (MB) used for message queuing
DOM	MCSODOM	DOM	NORMAL (see attribute)	Whether the console receives delete operator messages (DOM)
Extended MCS console key	MCSOKEY	KEY	NONE	1- to 8-byte name used on the DISPLAY CONSOLES command to identify extended MCS consoles
Automated messages	MCSOAUTY MCSOAUTN	AUTO	NO	Whether the extended console receives messages that are eligible for automation
Hardcopy	MCSOHDCY MCSOHDCN	HC	NO	Whether the extended console receives the hardcopy message set
Receive messages directed to console id zero	MCSOINT	INTIDS	N	Whether the console receives messages directed to console id zero.
Receive messages directed to unknown console ids	MCSOUNKN	UNKNIDS	N	Whether the console receives messages directed to unknown console ids, such as one-byte id.

The attributes have the following meanings:

Authority

Specifies the command authority for the console. MVS determines command authority by command groups defined as options for AUTH as follows:

INFO Informational commands

SYS System control commands

IO I/O control commands

CONS Console control commands

ALL Informational, system, I/O control, and console control commands

MASTER
Master authority commands

INFO is the lowest command authority. SYS, IO, CONS, and ALL are equivalent in authority. MASTER is the highest command authority.

Command group authority includes commands from groups with lower authority. For example, AUTH(SYS) allows the user to issue informational commands (INFO) as well as system control commands from the console. AUTH(INFO) is the default.

Routing codes
Specifies the routing codes (1 - 128) in effect for the console. MVS directs messages with the defined routing codes to the console. You can also specify:

ROUTCODE(ALL)
which sends all messages defined by routing codes to the console, or

ROUTCODE(NONE)
which doesn't use routing codes as criteria for routing messages to the console, and is the default.

Message level
Specifies the message level for the console, which indicates the type of message to be sent to the console. MVS distinguishes between kinds of message levels defined as options for LEVEL as follows:

R Write-to-operator with reply (WTOR) messages, which might demand an immediate reply.

I System failure and immediate action messages (descriptor codes 1 and 2), which indicate a serious error or that a task is awaiting a requested operator action.

CE Critical eventual action messages (descriptor code 11), which indicate that an eventual action of critical importance is requested on the part of the operator.

E Eventual action messages (descriptor code 3), which request an eventual action that does not require immediate operator attention.

NB Broadcast messages, which are messages normally sent to every active console regardless of the routing code you assigned to the console. Specifying LEVEL(NB) indicates that the console is not to receive broadcast messages.

IN Informational messages, which generally indicate system status. Most messages are informational.

ALL All messages, which indicate that all messages, including broadcast messages, appear on the console and is the default.

Message form
This parameter is necessary only if you are coding a presentation service and want to honor MFORM requests. If an MFORM value has been specified for this extended MCS console, you can find the value in the MCSCSA. The MCSCSA reflects the values specified in MCSOP. The system does not add a system name, job name, or time stamp into the message text. You can find those values in the MDB, if you want to include them as part of the message that your program presents.

Message scope

Identifies the system or systems in a sysplex from which the specified console will receive unsolicited messages.

The default is *ALL, which indicates that messages from the local system as well as all other systems in the sysplex appear on the console.

Command scope

Defines the system in a sysplex where you want to send commands entered on this console for processing. The default is an asterisk (*), which indicates that commands entered on the console are processed on the local system where the console is defined.

Monitor information

Specifies that you want messages sent to this console when system events, such as a job start or end or a TSO user logon or logoff, occur. MVS distinguishes among the following information, which you can specify as options for the MONITOR request:

JOBNAME

Specifies that the name of the job is given in job status messages whenever a job starts or ends.

STATUS

Specifies that data set names and volume serial numbers are given in status messages whenever data sets are freed.

SESS Specifies that the time sharing option extensions (TSO/E) user ID is given in session status messages whenever the TSO/E session begins and ends.

With JOBNAME or SESS, you can add a time stamp (-T).

Log command responses

Specifies whether the system logs messages that are responses to commands directed to the console. SYSTEM indicates that the value or default for HARDCOPY CMDLEVEL in CONSOLxx determines whether the system logs the command responses for the console, and is the default. NO indicates that the system does not log command responses for the console.

Storage

Specifies the limit of storage in megabytes used for message queuing. The maximum is 2048 megabytes. When the console is activated, the system sets the limit. 1 megabyte provides storage for about 250 message lines, and is the default.

DOM

Specifies whether the console receives delete-operator messages (DOMs). NORMAL indicates that the system attempts to queue all appropriate DOMs to the console, which is the default. ALL indicates that MVS queues all DOMs in the sysplex to the console. The application program that activates the console must indicate which DOMs it wants to receive based on its handling of held messages (that is, action messages and WTORs). If the MCSOPER ACTIVATE request specifies MSGDLVRY=NONE, the DOM attribute is forced to DOM=NONE.

Extended console key

Specifies a 1- to 8-byte character name used in the DISPLAY CONSOLES,KEY=keyvalue command. DISPLAY CONSOLES,KEY=keyvalue displays information for all consoles with that key. Thus, you can define a key that operators can use in the command to display information about all extended MCS consoles in the system or sysplex. The default value is NONE. For more information on the DISPLAY command, see *z/OS MVS System Commands*.

Automated messages
Specifies whether the console is to receive messages that are eligible for automation. Messages are flagged as automatable by specifying AUTO(YES/token) on the MPFLSTxx member of SYS1.PARMLIB. See *z/OS MVS Initialization and Tuning Reference* for more information on the message processing facility (MPF). Options are YES, which means the console receives messages that are eligible for automation, or NO, which means the console does not receive messages that are eligible for automation. NO is the default.

Hardcopy
Specifies whether the console is to receive the full hardcopy message set. Options are YES, which means the console receives the hardcopy message set, or NO, which means the console does not receive the hardcopy message set. Any route codes specified for a console do not apply for hardcopy messages, so users should be aware that the console will receive all hardcopy messages, regardless of their specific route code, when this option is set to YES. NO is the default. See *z/OS MVS Planning: Operations* for more information about the hardcopy message set.

Receive messages directed to console id zero
Whether the specified console can receive messages that are directed to console id zero. These messages are usually the command responses for internally issued commands.

Y The specified console is to receive these messages.

N The specified console is not to receive these messages. This is the default value.

Receive messages directed to unknown console ids
Whether the specified console can receive messages that are directed to "unknown" console ids. These ids are one-byte ids which the system cannot resolve.

Y The specified console is to receive these messages.

N The specified console is not to receive these messages. This is the default value.

Storing messages directed to an extended MCS console

The system stores messages differently, depending on how you will retrieve them. To retrieve a stored message, do the following:

1. Issue the MCSOPMSG macro to obtain the address of the MDB, which is where the system stores messages for an extended MCS console.
2. Use assembler instructions to locate and extract the message from the MDB.

MCSOPER provides the following options for delivery of messages:

- If you want MCSOPER to queue messages to and extract messages from the storage area on a first-in first-out basis, specify MSGDLVRY=FIFO on MCSOPER.
- If you want to use the search arguments CMDRESP, CART, and MASK when issuing the MCSOPMSG macro with the REQUEST=GETMSG parameter, specify MSGDLVRY=SEARCH on MCSOPER.
- If you do not want to receive messages on a console, specify MSGDLVRY=NONE on MCSOPER. MSGDLVRY=NONE requires that you

retrieve messages from either the subsystem interface or an MPF exit, rather than from the message data space. MSGDLVRY=NONE is useful if you want to issue commands from a console but do not want to receive messages on a console. The console ID identifies the particular console's messages on the subsystem interface or MPF exit. For more information on MPF exits, see *z/OS MVS Installation Exits*.

Controlling message traffic directed to an extended MCS console

This information explains how to control the number of messages queued to an extended console and the size of auxiliary storage that an extended console can use for the message data space.

- The STORAGE field in MCSOP specifies the size of the message data space created for the extended console. The range of the STORAGE field is from 1 to 2048, where 1 represents 1 megabyte of data space storage and 2048 represents 2048 megabytes, or 2 gigabytes, of data space storage. The default for STORAGE is 1. The number of messages going to your data space is limited by storage size. If you want to further control the number of messages going to your data space, use the QLIMIT parameter.
- The QLIMIT parameter on the MCSOPER macro specifies the maximum number of messages queued to an extended MCS console. The value can range from 1 to 2147483647 (2 billion). The default for QLIMIT is 2147483647. Taking the default might cause you to run out of storage in the message data space.
- The ALERTPCT parameter on the MCSOPER macro specifies a percentage of the QLIMIT value. When the number of messages queued to an extended MCS console exceeds the number represented by this percentage, the system alerts the extended MCS console by posting an ECB, identified by the ALERTECB parameter. For example, if the QLIMIT value is 2000, and the ALERTPCT value is 50, the system alerts the extended MCS console user when the number of messages in the message data space exceeds 50%, or 1000 messages, of the QLIMIT.

Receiving messages and command responses, and issuing commands

Once your extended console is activated, you can use it to perform the following functions:

- Receive unsolicited messages
- Receive command responses
- Receive the hardcopy message set
- Issue commands.

Receiving messages and command responses

Messages are sent to an extended MCS console when the system issues the WTO and WTOR macros. A message data block (MDB) contains the message delivered to the extended console. If the message is too long for one MDB, the MDB prefix area contains a pointer to the next MDB, where the message text continues. If multiple messages are sent to an extended MCS console, they will be placed into separate MDBs, requiring the issuance of separate MCSOPMSG macros to retrieve them. You can code a loop around the MCSOPMSG invocation so that the macro can retrieve messages currently queued to the extended MCS console. Exit the loop when there are no more messages (MCSOPMSG RC=8,RSN=0).

Once the message ECB of the extended MCS console has been posted, you can receive messages, and free storage for new messages by issuing the MCSOPMSG macro with the REQUEST=GETMSG option. You can receive messages by specifying certain message types. The CMDRESP option of MCSOPMSG allows you to retrieve the following messages:

- A command response (CMDRESP=YES)
- An unsolicited message or DOM (CMDRESP=NO).

If you choose to receive a command response, you can further specify the message command and response token (CART), which associates a response with a command. You can also specify a CART with a mask. Both the CART and mask are user-defined values originally specified on the MGCRE macro. A CART allows you to specify your selection criteria when retrieving command responses. A mask is a value that is compared to a CART with the logical AND instruction to broaden that selection criteria. For example, if the CART value is STR10001, MCSOPMSG searches for messages only with a CART value of STR10001. If the mask is X'FFFFFFFF00000000', MCSOPMSG uses the logical AND instruction to compare the two values and use the result as the search criteria. In this case, MCSOPMSG retrieves any command responses having STR1 as the first four characters of the CART.

If you do not specify CMDRESP, MCSOPMSG retrieves the next message in the queue on a first-in first-out basis.

If processing is successful, MCSOPMSG returns an address in GPR1 and an ALET in AR1; the GPR1/AR1 pair contains the address of the MDB. User applications are expected to copy the MDB from the message data space. On the next invocation of MCSOPMSG, the MDB in the message data space is deleted and returned back to the system, and either the next MDB will be returned to the caller, or a return and reason code will be given indicating that no more MDBs are currently queued.

You can also specify how the extended MCS console receives messages by specifying the MSGDLVRY parameter on the MCSOPER macro. MSGDLVRY enables you to request messages on a first-in-first-out basis (MSGDLVRY=FIFO) or by search arguments (MSGDLVRY=SEARCH) that you specify on MCSOPMSG. You can also request that no messages be queued to the extended MCS console (MSGDLVRY=NONE).

You need to understand the structure of the MDB to retrieve the message from it.

What is the message data block (MDB): The MDB is a data area that contains either a message or a DOM directed to an extended MCS console, and any information related to that message. It also contains a prefix area mapped by mapping macro IEAVG132. This area points to any subsequent MDBs for a given message. If there are no subsequent MDBs, this area contains a zero.

Messages reside in MDBs, which are put into a data space created when a program issues the MCSOPER macro to establish itself as an extended MCS console. When a message is sent to an extended MCS console, the system places the message in an MDB, where the message stays until the extended MCS console program issues the MCSOPMSG macro to locate the message.

If a DOM is directed to a previously issued message, the system creates a separate MDB for the DOM. The MDBGDOM field in the MDB identifies the MDB as a DOM MDB. The MDBDOMFL field indicates whether the console specified DOM(NORMAL) or DOM(ALL). If the console specified DOM(NORMAL), MDBGMID in the DOM MDB matches the same field in the message MDB. If the console specified DOM(ALL), the DOM field values in the control program object indicate which message or messages this DOM is to delete. A single DOM MDB can delete more than one message.

Figure 28 illustrates the structure of an MDB for a message that requires more than one MDB.

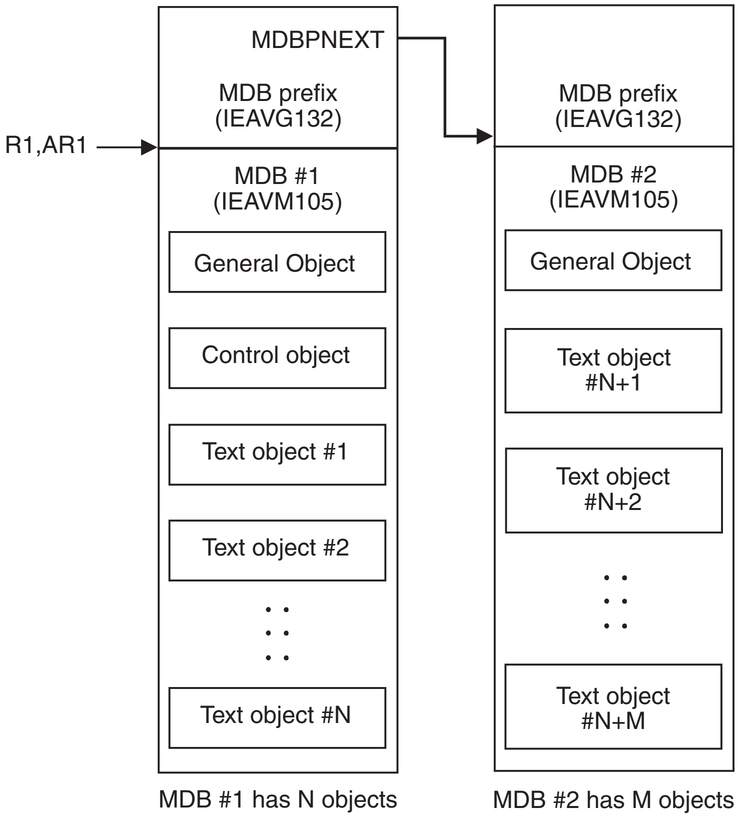


Figure 28. MDB Structure

The MDB generally contains three objects, each beginning with the length and MDB type specific to each object. To locate the objects, use the address of the MDB returned by MCSOPMSG. The object type identifies the object as a general, control, or text object. Within each object is a field that contains the length of the particular object. MDBGLEN contains the length of the general object, MDBCLEN contains the length of the control program object, and MDBTLEN contains the length of the text object.

To find the first object, add the length of the header to the MDB address. The object type will tell you whether it is a general, control, or text object. To find the end of the first object, add the length of that object to the end of the header.

You can find subsequent objects by identifying the end of the previous object, and adding the length of the next one. Compare the end value to the end of the MDB length to ensure that you are still looking within the MDB.

You can find the end of the MDB by adding the total length of the MDB, which is contained in the first field of the header, to the address of the MDB.

The three MDB objects are:

- The general object contains information about the message or DOM. It contains a message ID, time stamp, and indication of whether the MDB holds a WTO, WTOR, or a DOM. Each MDB has one general object.
- The control program object contains information about the message or DOM that is specific to MVS. If the message is a WTO or WTOR, the control program object contains routing codes, message level, job name, and other message information. If the MDB contains a DOM, the control program object contains the ASID, jobstep TCB, or system ID that deleted the message. Each MDB has one control program object.
- The text object contains the message text, which is variable in length. Each line of a message is represented by one text object. If the MDB cannot hold all text objects for a specific message, any remaining text objects will be placed in one or more additional MDBs. The MDB prefix points to the next MDB. That MDB will contain only the general object and the remaining text objects. If multiple messages are delivered, each message will be placed in a separate MDB, which must be retrieved via MCSOPMSG. These separate MDBs may, as before, consist of one MDB, or several MDBs chained together.

The general objects and the control program objects appear anywhere in the first MDB, but text objects are sorted in order within and among MDBs.

If the system sends a DOM to an extended MCS console, the MDB contains only two objects: the general object and the control program object.

Receiving the hardcopy message set

The hardcopy message set represents those messages that can be queued to the system log or OPERLOG. See *z/OS MVS Planning: Operations* for information about controlling and modifying the hardcopy message set definition.

Extended MCS consoles can also receive the hardcopy message set. To request that an extended MCS console receive the hardcopy message set, issue the MCSOPER macro with REQUEST=ACTIVATE, and specify the HARDCOPY attribute in the OPERPARM parameter list. “Specifying console attributes” on page 145 describes how to specify operator parameters.

You can also request that an extended MCS console receive the full hardcopy message set from all the systems in a sysplex, by specifying the MSCOPE=*ALL console attribute. Any route codes specified for a console do not apply for hardcopy messages, so users should be aware that the console will receive all hardcopy messages, regardless of their specific route code. Note that this might cause performance problems. The default for MSCOPE in a sysplex environment is ALL systems.

Issuing commands

To issue commands from an extended MCS console, issue the MGCRE macro with the CONSID or CONSNAME parameter. Specify a 4-byte console ID on the CONSID parameter or a 2- through 8-byte console name on the CONSNAME parameter. MCSOPER returns the CONSID parameter when you activate the extended MCS console. CONSNAME must be the same value you specified on the NAME parameter of MCSOPER. This console ID or name identifies the console that will receive responses to the commands you issue with MGCRE.

NOT Programming Interface information

The CMDFLAG=TSO option allows TSO to use MGCRE instead of MGCR. CMDFLAG=TSO causes the CONSID value in the parameter list to be treated as a TSO identifier, instead of a console ID.

Note: This option allows TSO to use MGCRE instead of MGCR.

End of NOT Programming Interface information

Specify a value for the CART parameter if you want to identify the response message for a particular command, or if you want to retrieve response messages with the MCSOPMSG macro based on CARTs or masks.

For more information on the MGCRE macro, see “Issuing operator commands from a program (MGCRE macro)” on page 137.

What to do if message queuing stops

When any of the following conditions occur, message queuing stops, and the system posts an ECB.

- No more storage remains in the message data space. The size of the storage area was specified in the STORAGE field of MCSOP.
- The maximum number of messages was reached in the message data space. The maximum number was specified on the QLIMIT parameter of the MCSOPER macro.

The ALERTECB parameter on the MCSOPER macro specifies the ECB to be posted when message queuing stops.

When MCSOPER posts the ECB, check the MCS console status area, MCSCSA to determine why queuing has stopped. If the message data space is full, the MCSCMLIM field contains a 1. If the maximum number of messages has been reached, the MCSCDLIM field contains a 1. In either case, to start message queuing again, retrieve messages from the MDB by issuing the MCSOPMSG macro.

Issue the MCSOPMSG macro with the REQUEST=GETMSG parameter a sufficient number of times to retrieve enough messages to meet the QRESUME percentage specified on the MCSOPER macro. For example, if on MCSOPER you defined QLIMIT as 1000 messages, and QRESUME at 50%, queuing stops when the message storage area holds 1000 messages and the system is queuing one more message into the area. You can restart queuing in one of two ways:

- Issue MCSOPMSG REQUEST=GETMSG 500 times so that the area holds 500 messages, or 50% of 1000. Queuing automatically resumes when the area holds 50% of the QLIMIT.
- Issue MCSOPMSG REQUEST=GETMSG any number of times to reduce the number of queued messages, then issue MCSOPMSG REQUEST=RESUME to restart queuing. Queuing will continue until you again hit the QLIMIT.

The system does not resume queuing if queuing stopped because the message data space reached the storage limit as defined for the storage attribute in MCSOP, a security product's user segment, or the system default.

Occasionally, the ALERTECB can be posted for one of the following conditions as well:

- An internal queuing error has occurred and message queuing has stopped. The MCSCSA internal error field will contain a 1. Issue MCSOPMSG REQUEST=RESUME to restart queuing.

By issuing MCSOPMSG REQUEST=RESUME, you can resume queuing to the message data space after any abend that suspends system message queuing, or after the failure of recovery routines invoked. Some or all messages currently queued might be lost. REQUEST=RESUME enables you to resume queuing without having to retrieve enough messages to reach the QRESUME percentage.

Deactivating extended MCS consoles

To deactivate an extended MCS console, issue the MCSOPER macro with the REQUEST=DEACTIVATE option.

There are certain instances where deactivation will be performed automatically by the system. For example: when the address space ends due to memory termination, or when a system in a sysplex is being partitioned.

Switching to another console

Through MCSOPER with REQUEST=DEACTIVATE and ABTERM=YES, you can:

- Switch processing from an extended MCS console to another active MCS or extended MCS console
- Deactivate an active console.

Switching an extended MCS console to an alternate console allows processing to continue without interruption, and is useful if a program representing an extended MCS console abnormally ends, and your installation needs to have its processing taken over by another console. You must specify the console ID of the console to be deactivated; for example:

```
MSCOPER REQUEST=DEACTIVATE,CONSID=(R4),ABTERM=YES
```

Before deactivating a console, you must define a valid alternate group for the console through operator attributes. See “Specifying console attributes” on page 145 for more information on operator attributes.

Removing extended MCS consoles

With z/OS V1R7, you can remove the definition of an unneeded EMCS console by using the sample program IEARELEC, which is shipped in SYS1.SAMPLIB. See *z/OS MVS Planning: Operations* for a description of this program and how to it. Removing these console definitions allows you to reuse the console ID assigned to the EMCS console, although IBM continues to recommend using a console name rather than a console ID for console communications.

An EMCS console can be deactivated on a downlevel system, but can only be removed on a z/OS V1R7 system after it has been deactivated. Once removed, the console ID can be reused by activating an EMCS console on a V1R7 system.

Example of managing an extended MCS console session

The following example shows an overview of how an application might use MVS services to set up and use an extended MCS console. Additional processing that a typical application might do, such as process a command response, is not shown but is represented by three stacked dots within the example. The example shows an application that:

- Activates an extended MCS console

- Issues a command from the console
- Waits until the system queues a message, the command response, to the console
- Receives the message from the console
- Deactivates the console.

```
*****************************************
* 1 - Activate an extended MCS console  *
*****************************************

               MCSOPER   REQUEST=ACTIVATE
                         NAME=CONS1              CONSOLE NAME
                         CONSID=(2)              CONSOLE ID
                         TERMNAME=(4)            TERMINAL NAME
                         MCSCSA=(3)              CONSOLE STATUS AREA
                         MCSCSAA=(3)             ALET FOR CONSOLE STATUS AREA
                         MSGECB=MESSAGE_ECB      MESSAGE ECB
⋮

****************************************
* 2 - Issue a command from the console *
****************************************

               XC        TEXTAREA,TEXTAREA       CLEAR THE COMMAND AREA
               L         R1,L'CMD                OBTAIN THE LENGTH OF THE COMMAND
               STH       R1,TEXTLEN              SAVE THE LENGTH IN THE COMMAND AREA
               MVC       TEXTCMD(L'CMD),CMD      PUT THE COMMAND IN THE COMMAND AREA
               MGCRE     TEXT=TEXTAREA           TEXTAREA CONTAINS THE COMMAND
                         CONSID=(2)              CONSOLE ID
                         CART=USER_DEF_CART      COMMAND/RESPONSE TOKEN
                         MF=(E,LISTADDR)         ADDRESS OF PARAMETER LIST
⋮

               DS        0F
TEXTAREA       DS        0CL128                  COMMAND AREA
TEXTLEN        DS        H                       LENGTH OF COMMAND
TEXTCMD        DS        CL126                   THE COMMAND
CMD            DC        C'DISPLAY EMCS EXAMPLE COMMAND'
⋮

*********************************************************
* 3 - Wait for the system to queue the command response *
*     to the console                                    *
*********************************************************
               WAIT      ECB=MESSAGE_ECB         WAIT FOR COMMAND RESPONSE
⋮
************************************************************
* 4 - Retrieve the command response so it can be processed *
************************************************************
               MCSOPMSG  REQUEST=GETMSG
                         CONSID=(2)              CONSOLE ID

⋮

*******************************************
* 5 - Deactivate the extended MCS console *
*******************************************
               MCSOPER   REQUEST=DEACTIVATE
                         CONSID=(2)              CONSOLE ID

⋮
MESSAGE_ECB    DS    F
USER_DEF_CART  DS    CL8
⋮
```

Figure 29. Managing an Extended MCS Console Session Overview

The following description provides additional detail about the example. The numbers in each of the following steps correspond to the numbers in the comment blocks above.

1. MCSOPER activates the extended MCS console named CONS1. The system returns the following values to the application:
   - In register 2, the address of the 4-byte console identifier
   - In register 3, the address of the console status area
   - In AR3, the ALET for the address space or data space that contains the console status area.

   The MSGECB parameter identifies the ECB that the system is to post when there is a message queued for the extended MCS console.
2. MGCRE issues a command from the extended MCS console. The CART parameter identifies a user-defined token that the command processor is to return with any response related to the command being issued.
3. In this example, the application cannot continue processing until it receives a command response. Therefore, the application waits on the ECB. After the system issues the command response the ECB gets posted and the application can continue processing.
4. MCSOPMSG retrieves the message that is queued to the extended MCS console and places it into a message data block (MDB). In this case, the message is the command response. The MDB also contains the command response token the application provided on the CART parameter of the MGCRE macro. The system returns the address of the MDB in register 1 and the ALET for the address space or data space where the MDB is located in AR1. For more information on the MDB, see "What is the message data block (MDB)" on page 151. When an application has issued multiple commands and expects command responses, the command response token enables the application to associate each response with the corresponding command.
5. The application issues MCSOPER REQUEST=DEACTIVATE to deactivate the extended MCS console.

SYS1.SAMPLIB contains an additional example of an extended MCS console application. The example is in IEAEXMCS.

Chapter 8. Listening for system events

The event notification facility (ENF) allows an authorized program to listen for the occurrence of a specific system event. For example, a program could use ENF to listen for a software record being written to the logrec data set. When the system writes the record to the logrec data set, control passes to a specified listener user exit routine. For a list of the system events that an authorized program can listen for, see "ENF event codes and meanings" on page 168.

To listen for a system event, use the ENFREQ macro with the LISTEN option to specify:

- The specific event for which you would like to listen
- The listener user exit routine that is to receive control after the specified event occurs
- Address of an ENF Listener filter block (for signals that support filter blocks)
- An optional DTOKEN to identify your listen request when ENF returns control
- Any parameters that you would like to pass
- Whether the listener user exit routine should end when the listener's task or address space ends
- Whether you would like the listener user exit routine to run in the listener's address space (SRB mode).

In addition to issuing ENFREQ, you must also install your listener user exit routine in common storage if the listener user exit routine is not an SRB exit.

To stop listening for a system event, use ENFREQ with the DELETE option to specify:

- The specific event for which your program was listening
- The DTOKEN that you received on the original listen request.

The information describe how to establish a listen request and code a listener user exit routine. For the complete syntax of the ENFREQ macro, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Establishing a listen request

Each event is identified by a decimal event code. To specify which event you would like to listen for, set the CODE parameter on the ENFREQ macro equal to the desired event code. For a list of the event codes and their corresponding events, see "ENF event codes and meanings" on page 168.

Qualifying events

For certain types of events, you can use the *QUAL* and the *QMASK* parameters on the ENFREQ macro to more specifically define the event for which you would like to listen. For example, if you were listening for event code 37 without using the QUAL and QMASK parameters, you would receive notification when any SMF accounting-related event occurred. By using the QUAL and QMASK parameters, however, you can specify that you want to receive notification only when certain SMF accounting-related events occur. For instance, you can request notification only when SMF is initialized.

The QMASK parameter, in combination with the QUAL parameter, specifies the bytes of the four-byte qualifier that ENF uses to further define the event. The listener user exit routine receives control only when a system event occurs that matches the characteristics specified by the QMASK bytes of the QUAL field. The default for QMASK is QMASK=NONE.

For example, if QMASK=BYTE1, the listener user exit routine receives control when an event with characteristics described by the first byte in the qualifier occurs. Because QMASK=BYTE1, ENF ignores information in bytes 2 through 4.

You can specify the following possible QMASK values:

BYTE1
First byte

BYTE2
Second byte

BYTE3
Third byte

BYTE4
Fourth byte

ALL All four bytes

NONE
No bytes

If you specify ALL or NONE, do not code any other QMASK values. If you do not specify any QMASK values, the default is NONE.

The qualifiers that are valid depend on the system event for which you are listening. For a list of QUAL values see “ENF event codes and meanings” on page 168.

QUAL example

The following three figures explain how an authorized program can use QUAL and QMASK with event code 36 to more specifically define the type of event code 36 for which the program would like to receive notification from ENF. Event code 36 specifies notification when a software record is written to the logrec data set or logrec log stream. To limit the records that the authorized program receives notification about, the authorized program specifies one of the qualifier values listed in “ENF event codes and meanings” on page 168. Assume that the following have been defined for use by the program in these qualifier examples:

```
ENF36PGMCHECK    EQU   X'42'   Indicates a program check occurred
ENF36MACHCHECK   EQU   X'48'   Indicates a machine check occurred
```

In Figure 30 on page 161, the system writes a record when a program check occurs and when a machine check occurs. Because the listening program does not specify any QUAL or QMASK values, ENF passes control to the listener user exit routine when the system writes either of the software records to the logrec data set or logrec log stream.

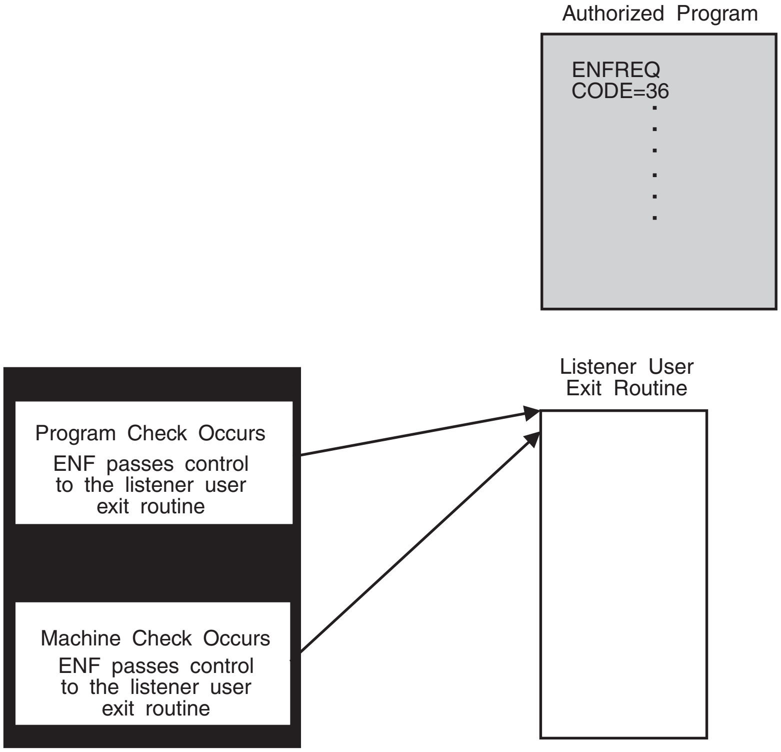


Figure 30. Authorized Program without QUAL and QMASK

To specifically define which ENF signals the program should receive, you must code QUAL and QMASK. Figure 31 on page 162 shows an example of how the QMASK bytes of a listener's qualifier must match the signaller's qualifier. In this example, a listener uses ENF36PGMCHECK, the program check qualifier for event code 36, to request notification when a program check occurs. The listening program sets QMASK to BYTE4, because the fourth byte contains the information that specifically identifies a program check. When a program check occurs, the fourth byte of the signaller's qualifier equals the listener's, and ENF notifies the listener that a program check occurred.

Note that not all event codes use a QMASK of BYTE4. The QMASK varies with the qualifiers for a given event code.

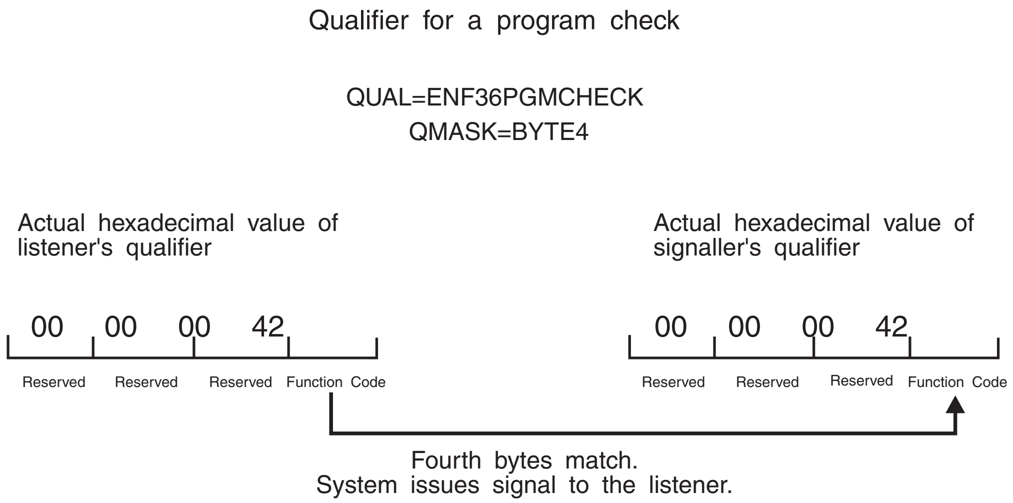

Qualifier for a program check
QUAL=ENF36PGMCHECK
QMASK=BYTE4
Actual hexadecimal value of listener's qualifier
Actual hexadecimal value of signaller's qualifier
00
00
00
42
Reserved
Reserved
Reserved
Function Code
00
00
00
42
Reserved
Reserved
Reserved
Function Code
Fourth bytes match.
System issues signal to the listener.

Figure 31. Qualifier for a Program Check

Figure 32 on page 163 shows two authorized programs that are both listening for event code 36, but they have different QUAL values. Authorized program 1 has a QUAL of ENF36PGMCHECK (which corresponds to a software record for a program check) and a QMASK of BYTE4. Authorized program 2 has a QUAL of ENF36MACHCHECK (which corresponds to a software record for a machine check) and a QMASK of BYTE4.

A QMASK value of BYTE4 tells ENF that it should only use the fourth byte of the qualifier to determine the listener user exit routine that gets control. Note that the authorized programs in this example could use a QMASK of ALL. Because there is no information contained in the first three bytes, ENF would only use the fourth byte. However, because the first three bytes might contain data, IBM recommends that you use a QMASK value of BYTE4.

The listener user exit for authorized program 1 receives control only when a software record is written for a program check. The listener user exit for authorized program 2, on the other hand, receives control only when a software record is written for a machine check.

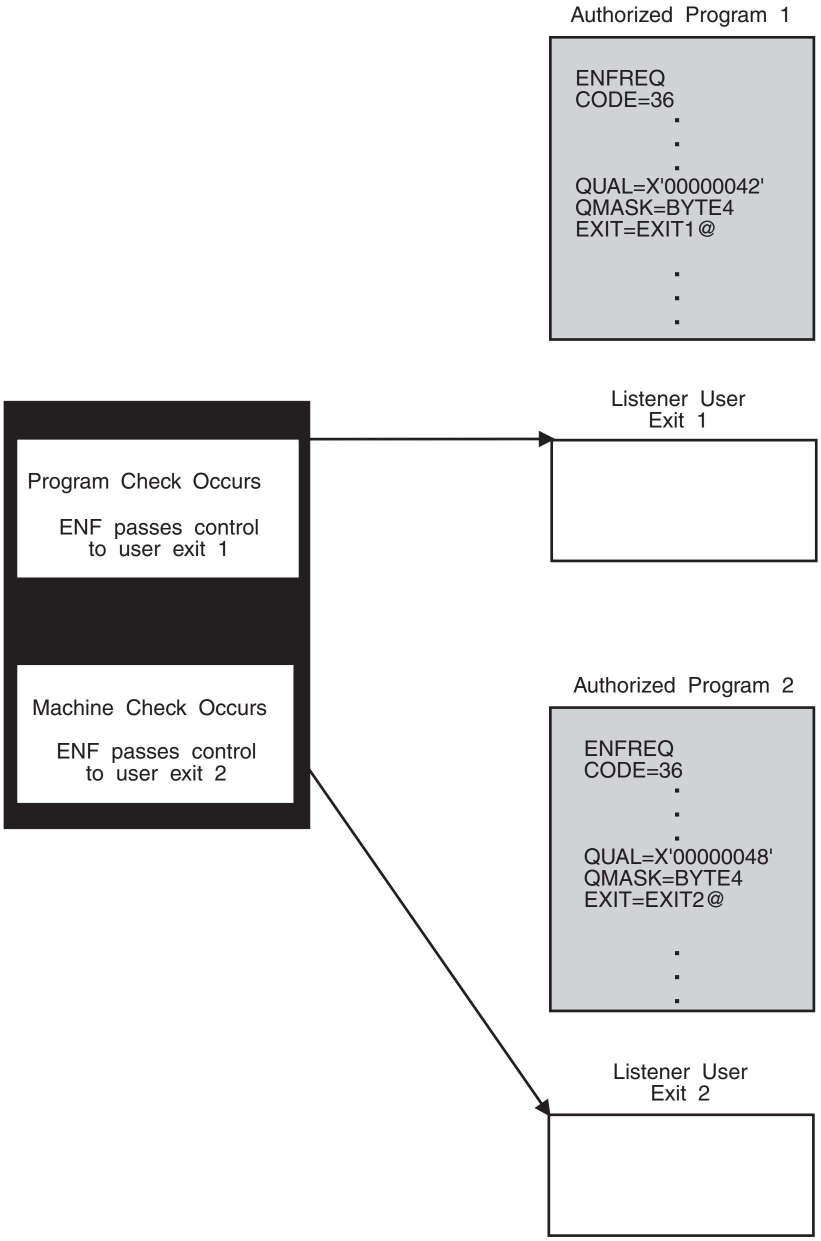


Figure 32. Authorized Programs with QUAL and QMASK

nn filtering events

Some types of event support the FLTRBLK parameter on the ENFREQ macro to identify the address of an optional Enf Listener Filter block. The filter block data is specific to the signal type and identify further criteria to be used to determine if the signal should be delivered to the Enf listener. ENF Event code 51 supports filter blocks that allow further filtering of GRS contention data signals to ones that involve specific and patterned qnames and rnames that might be of interest to that Enf Listener. The Enf 51 Listener uses the FltrBlk keyword to identify the address of the filter block for what signals are to be filtered further by the supporting Enf 51 signaller beyond that of signals matching just the Qual and QMask keywords.

Coding the listener user exit routine

After the event for which you are listening occurs, control passes to the listener user exit routine you specify on either the EXIT parameter or the SRBEXIT parameter of the ENFREQ macro. The exit receives control in the event issuer's address space. If you want the listener user exit routine to receive control in SRB mode in the address space that issued the listen request, specify SRBEXIT. For example, if the listener user exit routine needs to address the requestor's private storage, use SRBEXIT. SRBEXIT is valid only with certain event codes. See “ENF event codes and meanings” on page 168 for a list of all codes and their meanings.

If, for documentation purposes, you want to specify names for the listener user exit routine and the exit's establisher, use the EXITNME and ESTBNME parameters on ENFREQ. Although these parameters are optional, IBM recommends that you use them.

The listen exit is coded to run in either task or SRB mode, depending on the parameter you specify. If you specify SRBEXIT, it can only run in SRB mode. If you specify EXIT, it is able to run in both task and SRB modes. The Listen exit does not impose any restriction on the signaller.

Installing the Exit: The listener user exit routine must reside in common storage if the exit routine is not an SRB exit. To install the listener exit routine in common storage, perform the following steps:

- (For a subsequent IPL) Link-edit the routine into a data set in the LPA concatenation or use SMP/E to place it there, or
- (For the current IPL) Use Dynamic LPA services to add the routine to LPA.

If the exit routine is an SRB exit, it can reside in either common or private storage.

Non-SRBEXIT routine

The following information is about standard listener user exit routines. For information about SRB listener user exit routines, see “SRBEXIT routine” on page 166.

Exit Routine Environment: The listener user exit receives control in the following environment:

Environmental factor	**Requirement**
Authorization:	Supervisor state and PSW key 0
Dispatchable Unit Mode:	Task or SRB
Cross memory mode:	PASN=HASN=SASN
AMODE:	Depends on the setting of the high-order bit of the field containing the exit routine address. (24-bit if the bit is off, 31-bit if the bit is on)
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held

Exit recovery: If your listener user exit routine acquires resources, such as storage or locks, provide a recovery routine for the exit.

If the listener user exit routine ends abnormally, control proceeds to the exit's recovery routine, if it provides one, or to a system recovery routine. The system recovery routine ends the notification request so that the exit routine that abended can no longer receive control. ENF continues, however, to send notification to the other listening programs.

The system will not end the notification request for an exit routine if a non-retryable abend, such as a CANCEL command, caused the listener user exit routine to end abnormally.

Listener user exit processing: The high-order bit of the listener user exit address specified on the ENFREQ macro request determines the AMODE in which the listener user exit routine receives control.

Programming considerations: The listener user exit routine must be reentrant.

Instructions and restrictions: Avoid such time-consuming processing as obtaining large amounts of storage through the GETMAIN macro, issuing WAITs or calling routines that issue the WAIT macro, and performing I/O operations. Also, avoid requests for the local lock and using multiple listener user exits. Avoid the use of dynamic allocation (SVC99) calls in "listen" exits that receive control in ASID 1; such calls can cause unpredictable results.

Because the listener user exit routine does not normally receive control in the same address space as the LISTEN request, the listener user exit routine cannot depend on the original address space always existing.

Entry specifications: The system passes information to the listener user exit routine in a parameter list that is mapped by the event code's mapping macro. For information about the parameter lists for each event code, see “ENF event codes and meanings” on page 168.

Registers at entry: On entry to the listener user exit routine, the registers contain the following information:

Register	Contents
GPR 0	Event code
GPR 1	Address of a 6-word structure. **Word** Contains **1** Address of the parameter list supplied by the system for this event code **2** Fullword of zeros **3** Fullword of data specified on the PARM keyword or the address of the listener parameter list specified on the PARM keyword **4** Fullword of zeros **5** Address of a parameter list mapped by the IEFENFSG macro **6** Fullword of zeros See “Passing parameters to a listener user exit routine” on page 167.
GPRs 2-12	Do not contain any information for use by the listener user exit.
GPR 13	Address of an 18-word save area
GPR 14	Return address
GPR 15	Entry point address of the listener user exit routine

Return specifications: The listener user exit routine does not need to set any return codes.

Registers at exit: Upon return from listener user exit routine processing, the registers must be:

Register	Contents
0, 1	The listener user exit does not have to place any information in this register, and does not have to restore its contents to what they were when the listener user exit routine received control.
2 - 13	The listener user exit must restore the contents to what they were when the exit received control.
14, 15	The listener user exit does not have to place any information in this register, and does not have to restore its contents to what they were when the listener user exit routine received control.

SRBEXIT routine

The following information is about SRB listener user exit routines. These exit routines are valid with only certain event codes. See “ENF event codes and meanings” on page 168 for a list of the codes that allow SRBEXIT.

The listener user SRBEXIT receives control in the following environment:

Environmental factor	**Requirement**
Authorization:	Supervisor state
Dispatchable unit mode:	SRB
Cross memory mode:	PASN=HASN=SASN
AMODE:	31-bit
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held

Exit recovery: If the SRB listener user exit routine ends abnormally, control proceeds to the exit's recovery routine, if it provides one, or to a system recovery routine. The system recovery routine ends the notification request so that the exit routine that abended can no longer receive control. ENF continues, however, to send notification to the other listening programs.

The system will not end the notification request for an exit routine if a non-retryable abend, such as a CANCEL command, caused the SRB listener user exit routine to end abnormally.

Instructions and restrictions: Avoid such time-consuming processing as obtaining large amounts of storage through the GETMAIN macro, issuing WAITs or issuing SVCs that issue the WAIT macro, and performing I/O operations. Also, avoid requests for the local lock and using multiple listener user exits.

Registers at entry: On entry to the ENF listener user exit routine, the registers contain the following information:

Register	Contents
GPR 0	Event code

Register	Contents
GPR 1	Address of a 6-word structure. **Word** **Contains** **1** Address of the parameter list supplied by the system for this event code. **2** Fullword of zeroes **3** Fullword of data specified on the PARM keyword or the address of the listener parameter list specified on the PARM keyword **4** Fullword of zeroes **5** Address of a parameter list mapped by the IEFENFSG macro **6** Fullword of zeroes See "Passing parameters to a listener user exit routine."
GPRs 2-12	Do not contain any information for use by the listener user exit.
GPR 13	Address of an 18-word save area
GPR 14	Return address
GPR 15	Entry point address of the listener user exit

Return specifications: The ENF listener user exit routine does not have to set any return codes.

The SRB listener user exit routine must return to the address in register 14.

Registers at exit: Upon return from listen exit processing, the registers must be:

Register	Contents
0, 1	The listener user exit does not have to place any information in this register, and does not have to restore its contents to what they were when the listener user exit routine received control.
2 - 13	The listener user exit must restore the contents to what they were when the exit received control.
14, 15	The listener user exit does not have to place any information in this register, and does not have to restore its contents to what they were when the listener user exit routine received control.

Passing parameters to a listener user exit routine

The listener user exit routine receives parameters in a six-word data structure that is pointed to by register 1 on entry to the listener user exit routine.

The six-word data structure contains parameters from one or both of the following sources:

- The system. The first fullword of the data structure contains the address of the parameter list associated with the ENF event code's mapping macro. See "ENF event codes and meanings" on page 168 for a list of the mapping macros.
- The ENF listener. The listening program can send parameters to the listener user exit routine by using the PARM keyword on the ENFREQ macro. PARM can be either a fullword of data or the address of a parameter list. PARM is passed to the listener user exit routine in the third word of the data structure.

Ending the listener user exit routine

Using the EOT and EOM parameters, you can specify on the ENFREQ macro whether the system should automatically delete the listener user exit routine when the listener's task ends (EOT) or when the address space ends (EOM). If you specify SRBEXIT then you must specify EOM=YES because the exit routine runs in the listener's address space and ends with the listener.

Only certain combinations of EOT and EOM are valid. The following chart shows which combinations are valid.

Table 20. Valid EOT and EOM parameter combinations

EOT	EOM	Exit type
EOT=YES	EOM=YES	EXIT or SRBEXIT
EOT=NO	EOM=YES	EXIT or SRBEXIT
EOT=NO	EOM=NO	EXIT

For example, to end the exit routine when the listener's task ends or the listener’s address space ends, specify EOT=YES and EOM=YES. To end the exit routine when the listener's address space ends, but not when the listener's task ends, specify EOM=YES, EOT=NO. For more information about the EOM and EOT keywords, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

ENF event codes and meanings

The following characteristics vary depending on the event for which you are listening.

Event code
Identifies the event

Qualifier
Further defines the specific event for which you would like to listen

Parameter list
Passes information about the event to the listener user exit

Exit type
Specifies the type of the listener user exit routine, which can be either EXIT or SRBEXIT

Cross-system capable
Specifies whether the exit is to receive signals from other systems in the sysplex

Table 21. ENF macro event codes

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
20	**Notes:** 1. The input save area and the information area (registers 13 and 1) point to areas above 16M. When specifying ENFREQ REQUEST=LISTEN, make sure that the exit routine (EXIT keyword) gets control in AMODE 31. 2. This exit only gets control in task mode in ASID 1. 3. Event 20 "listen" exits should avoid issuing dynamic allocation (SVC99) calls.	The defined QUAL values are: **Qualifier** **Information type** **x'80000004'** System information changed. Any program using the data returned by the CSRSI service should obtain the updated data.	Mapped by SIV1V2V3 DSECT within macro CSRSIIDF. This area contains the current information that would be returned by the CSRSI service when all data is requested (a request type of CSRSI_TYPE_ V1CPC_Machine plus CSRSI_TYPE_ V2CPC_LPAR plus CSRSI_TYPE_ V3CPC_VM), with the exception of the fields whose names begin with "SI00PCCA". If the SI00PCCA*xxx* fields are needed, the CSRSI service can be called. The SIV1V2V3 area is in 31-bit storage.	EXIT / NO
23	The system or an operator varied a device online. An operator can vary a device online by using the VARY command. For more information about the VARY command, see *z/OS MVS System Commands*.	Corresponds to the UCBTYP field in the UCB data area. The bytes in the qualifier correspond to the bytes in UCBTYP as follows: First byte = UCBDVCLS Second byte = UCBUNTYP Third byte = UCBTBYT2 Fourth byte = UCBTBYT1	Mapped by IEFEVARY	EXIT / NO
24	The system or an operator varied a device offline. An operator can vary a device offline by using the VARY command. For more information about the VARY command, see *z/OS MVS System Commands*.	Corresponds to the UCBTYP field in the UCB data area. The bytes in the qualifier correspond to the bytes in UCBTYP as follows: First byte = UCBDVCLS Second byte = UCBUNTYP Third byte = UCBTBYT2 Fourth byte = UCBTBYT1	Mapped by IEFEVARY	EXIT / NO
25	The system or an operator unloaded a DASD or tape volume. An operator can unload a DASD volume by issuing the VARY command. For more information about the VARY command, see *z/OS MVS System Commands*.	Corresponds to the UCBTYP field in the UCB data area. The bytes in the qualifier correspond to the bytes in UCBTYP as follows: First byte = UCBDVCLS Second byte = UCBUNTYP Third byte = UCBTBYT2 Fourth byte = UCBTBYT1	Mapped by IEZEUNLD	EXIT / NO
28	A dynamic device reconfiguration (DDR) swap occurred. A DDR swap moves or swaps a demountable volume from a failed device to another available device. For information about the SWAP command, which enables an operator to perform a DDR swap, see *z/OS MVS System Commands*.	None	8-byte parameter list. The first four bytes contain the address of the UCB for the device that was the source of the swap event. The second four bytes contain the address of the UCB for the device that was the target of the swap event.	EXIT / NO
29	The system or an operator placed a device in pending offline status. An operator can place a device in offline status by issuing the VARY command. For more information about this command, see *z/OS MVS System Commands*.	Corresponds to the UCBTYP field in the UCB data area. The bytes in the qualifier correspond to the bytes in UCBTYP as follows: First byte = UCBDVCLS Second byte = UCBUNTYP Third byte = UCBTBYT2 Fourth byte = UCBTBYT1	Mapped by IEFEVARY	EXIT / NO
30	The system or an operator placed a volume online so that it would be available for system use. An operator can place a volume online by issuing the VARY command. For more information about this command, see *z/OS MVS System Commands*.	Corresponds to the UCBTYP field in the UCB data area. The bytes in the qualifier correspond to the bytes in UCBTYP as follows: First byte = UCBDVCLS Second byte = UCBUNTYP Third byte = UCBTBYT2 Fourth byte = UCBTBYT1	Mapped by IEFEVARY	EXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
31	A configuration change that involves deleting a device or deleting a path to a device was requested or was rejected.	None	Mapped by IOSDDCCD	EXIT / NO
32	A configuration change was successful.	None	Mapped by IOSDDCCD	EXIT / NO
33	One of the following changes to the hardware configuration of a device occurred: • A device is added or deleted from the hardware configuration definition or a device is attached or detached with the VM ATTACH or DETACH command. The I/O subchannel corresponding to the device's UCB is connected or disconnected. • A device is made available because the channel path to the device is reestablished. • The description of a device is added, deleted, or changed. The self-description information is stored in a configuration data record (CDR). A change to a CDR is always a delete followed by an add. Use timestamps to determine the correct sequence. • The HyperPAV mode of operation for a logical control unit is changed. • A change in state has occurred for a PCIe device. • A device requires monitoring.	**BYTE 1** Device class (Byte 3 from UCBTYP) **BYTE 2** Reserved **BYTES 3-4** Qualifier number Each qualifier number designates a type of change, such as I/O subchannel change, device available, a configuration data record (CDR) change, or a HyperPav mode change. Along with each qualifier number is a qualifier number-dependent mapping in the IOSDDACH mapping macro, which designates fields specific to the type of change. The following ENF signal 33 subtypes are issued for PAV-alias devices: • DACHIO • DACHIORA • DACHCCDR • DACHPAV For each of these subtypes, if the signal applies to a device in the alternate subchannel set, the issuer will fill in a new field in the DACH subtype for the subchannel set identifier and change the subtype according to the information in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). ENF 33 is issued once for the logical control unit when its HyperPav mode is changed. The following ENF 33 subtype fields are updated to uniquely identify this event: • DACHDEVC='CU' • DACHTRAN='TRAN' • DACHQN=X'0008' • DACH_TRAN_CU=control unit that is changing • DACH_TRAN_MODE=target mode of operation For specific field definitions, see the IOSDDACH macro in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). The ENF signal 33 subtype DACHPCIE (X'0009') is issued for the change in state of a PCIe device. For this subtype, the following value is set in the DACHTYPE field by the issuer of the signal: DACHTYPEPCIE ('PCIE'). For this subtype, the DACHQN field is set to X'0009' (PCIE device event).	Mapped by IOSDDACH	EXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
33 (cont.)		For this subtype, the following data is supplied in the DACHQUALD field by the issuer of the signal: • DACH_PCIE_PFID (4 bytes): The PFID of PCIe device involved in the event. • DACH_PCIE_DEVID (2 bytes): The device ID of PCIe device involved in the event. • DACH_PCIE_VENDID (2 bytes): The vendor ID of PCIe device involved in the event. • DACH_PCIE_EVENT (1 byte): The device event code: 1 = The device is going online. 2 = The device is going offline. ENF signal 33 subtype DACHMONC (DACHTYPE = DACHMONC) is issued when a change in device monitoring is requested. This signal may be generated for secondary devices monitored for HyperSwap® configurations for which I/O operations may begin to be started. This signal may also be received when devices that were previously identified by ENF 33 subtype DACHMONC no longer require monitoring. Programs such as RMF may choose to monitor this ENF 33 signal to know when to begin collecting data for these devices which might otherwise see only insignificant amounts of I/O activity. For the DACHMONC subtype: • For a device that requires monitoring, the following fields are set: **DACH_IO_QUAL** Set to DACH_IO_QUAL_MONC_ON **DACH_IO_DEVN** Device number **DACH_IO_SSID** Subchannel set identifier **DACH_IO_DTYP** Contents of the UCBTYP field from the UCB **DACHUCBC** Device class **DACHQN** Either DACHIO or DACHIO_AS Each device receives a separate signal to begin monitoring. When monitoring is requested, the UCBCMONR bit is set on in the UCB. • When one or more devices no longer require monitoring, a single ENF 33 DACHTYPE = DACHMONC, DACHQN = DACHIO signal is given with DACH_IO_QUAL = DACH_IO_QUAL_MONC_OFF. This is typically done after a HyperSwap occurs or when a configuration is purged from the HyperSwap manager. When this signal is received, a UCBSCAN can be done to detect devices for which the UCBCMONR bit is no longer set on. DACH_IO_DEVN and DACH_IO_SSID are not used for this signal.		

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
35	One of the following XES or XCF events has occurred: • New coupling facility resources have become available on this system. Requests to connect with IXLCONN that previously failed might now succeed because of this new coupling facility resource. • A specific structure has become available for use. Requests to connect to the structure with IXLCONN that previously failed might now succeed because of this new coupling facility resource. • A system has joined the sysplex. The system name and ID are presented to the user. • A system has been partitioned from the sysplex. The system name and ID are presented to the user. • A CF definition with a SITE specified has been added or an existing CF SITE specification has changed. Note that the listener user exit routine for event code 35 can run in SRB mode.	None	Mapped by IXCYENF	EXIT or SRBEXIT / NO
36	The system wrote a record to the logrec data set or the logrec log stream. ENF passes to the listener user exit routine a parameter list containing the record information. For details about the contents of the parameter list, see IFBENF36 in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). Additional considerations for listeners of this code include the following: • The mapping does not indicate whether an IBM or non-IBM program caused the record to be written to logrec. • ENF does not suppress duplicate ENF signals sent to the listener. The listener must be aware of instances where a program loop causes the same software record to be recorded in logrec multiple times, thus causing ENF to issue duplicate signals. • ENF does not filter software records based on any criteria including ABEND codes.	The specific logrec record type value is used as the qualifier for each ENF event code 36 signal. Note that no signal is issued for record types X'9*x*'.	Mapped by IFBENF36	EXIT or SRBEXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
37	One of the following SMF accounting-related events occurred: • SMF was initialized • SMF ended • SMF INTVAL parameter changed • SMF SYNCVAL parameter changed • SMF interval expired • SMF interval sync processing disabled • SMF event driven interval occurred For information about these accounting-related events, see *z/OS MVS System Management Facilities (SMF)*.	**ENF37Q00** SMF address space was initialized. **ENF37Q01** SMF address space ended. **ENF37Q02** SMF INTVAL parameter changed. **ENF37Q03** SMF SYNCVAL parameter changed. **ENF37Q04** SMF interval expired. **ENF37Q05** SMF interval sync processing disabled. **ENF37Q06** SMF event driven interval occurred.	Mapped by IFAENF37	EXIT / NO
38	One of the following automatic restart manager events occurred: • A job or task started or was restarted, and has registered or reregistered as an element of the automatic restart manager. • An element notified the system that it is ready to accept work. • An element has deregistered with the automatic restart manager. • This system has acquired (or regained) access to the automatic restart management couple data set. Batch jobs and started tasks may now register as elements of the automatic restart manager. • An element has been deregistered with the automatic restart manager.	None	Mapped by IXCYAREN	SRBEXIT / NO
40	A JES2 subsystem either completed initialization or ended normally. (Note that ENF code 40 does not reflect situations in which JES2 abends.) ENF passes to the listener user exit routine a parameter list that identifies the JES2 subsystem. For details about the contents of the parameter list, see IEFENF40 in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).	**ENF40_INIT** A JES completed initialization. **ENF40_TERM** A JES ended normally	Mapped by IEFENF40	EXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
41	A workload management (WLM) event occurred. The following qualifiers for ENF code 41 are provided: **BYTE 1** **1** Policy change was initiated. **2** Policy change completed. **3** Policy change failed. **BYTE 2** Reserved. **BYTE 3** **1** Workload activity reporting failed and has begun recovery. **2** Workload activity reporting recovery was successful. **3** Workload activity reporting recovery was not successful. **BYTE 4** **1** WLM service definition was successfully installed.	**WLMENF11** A VARY WLM,POLICY command was issued. **WLMENF12** A VARY WLM,POLICY command completed. **WLMENF13** A VARY WLM,POLICY command failed. The new policy could not be activated on this system. **WLMENF31** WLM workload activity reporting failed and has begun recovery. **WLMENF32** WLM workload activity reporting recovery was successful. **WLMENF33** Workload activity reporting recovery was unsuccessful. **WLMENF41** Service definition was successfully installed.	Mapped by IWMRENF1	EXIT / NO
43	A new copy of workload management sampled address space information is available via IWMRQRY. Event code 43 is issued at the end of workload management's sampling interval so a listener can synchronize its sampling interval with workload management's interval.	None	Four byte parameter containing the length of the storage required to hold the information. A listener can pass this length to IWMRQRY in the ANSLEN parameter and save issuing IWMRQRY to determine the length.	EXIT / NO
44	A configuration change involving paths to a coupling facility has occurred.	None	Mapped by IXLYCFSE	EXIT / NO
45	The SMSVSAM server address space has been initialized or reinitialized after a failure. Any subsystem that lost connection to the service provider address space can now reconnect.	None	Mapped by IDAENF45	SRBEXIT / NO
46	z/OS UNIX System Services has been initialized or reinitialized.	None	None	EXIT / NO
47	DAE has detected that the threshold for completed or suppressed dumps, related to a particular symptom string, has been reached.	None	Mapped by ADYENF	EXIT / NO
48	A status change has occurred within system logger. The events issued by ENF 48 are issued to all systems in the sysplex. For a description of using ENF event 48 for system logger, see *z/OS MVS Programming: Authorized Assembler Services Guide*. For a description of the events mapped by the IXGENF macro, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).	None	Mapped by IXGENF	SRBEXIT / YES
49	The logrec output recording medium has been changed by the SETLOGRC command.	None	IFBNTASM	SRBEXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
51	One of the following types of GRS information: • Resource contention information • RNL change effects on user jobs • GRS mode change information Note that the listener user exit routine for event code 51 can run in SRB mode. Event code 51 can generate large numbers of events in short periods of time. The listener user exit routine for event code 51 must handle the volume of events. See *z/OS MVS Programming: Authorized Assembler Services Guide* for a description of system services to avoid when writing listener user exits. Supports Filter Block (FLKBLOCK) listners: Mapped by ISGYELF. The filter block reason codes for EnfReq RC=X'68' is in field ISGYELF_ReasonCode. The mapping also includes constants for the various values of the reason code.	The qualifier (QUAL parameter) has the following format: **BYTE 1** Type of signal information: **x'01'** Contention data **x'02'** RNL changes **x'03'** Mode changes **BYTE 2** Always x'00'. **BYTE 3** Varies with type of signal (value of BYTE1): **x'00'** Normal contention **x'01'** Waitless contention **BYTE 4** Varies with type of signal (value of BYTE1): **x'01'** Local events **x'02'** Global events **x'03'** Recovery events The defined QUAL values are: **Qualifier** **Information type** **x'01000000'** All Normal resource contention (excludes waitless) **x'01000001'** Normal Local resource contention **x'01000002'** Normal Global resource contention **x'01000003'** Normal Contention-related recovery information **x'01000100'** All Waitless resource contention **x'01000101'** Waitless Local resource contention **x'01000102'** Waitless Global resource contention **x'02000001'** User job suspended because of RNL change **x'02000002'** User job resumed following RNL change **x'0300*yyzz*'** GRS mode changes: *yy* Old mode *zz* New mode Values for *yy* and *zz* are those defined in IHAECVT for the ECVTGMOD field	Contention data: ISGE51CN RNL data: ISGE51RN Mode change data: None	EXIT or SRBEXIT / YES
52	A LNKLST set has been activated. A LNKLST set can be activated at IPL through a PROGxx LNKLST statement, or through SET PROG=xx or SETPROG LNKLST operator commands. For information about PROGxx, see *z/OS MVS Initialization and Tuning Reference*. For commands, see *z/OS MVS System Commands*.	None	Mapped by CSVDLENF	EXIT or SRBEXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
53	• A Sysplex Timer (ETR) configuration change occurred. • A change to the local time offset occurred.	None	8-byte parameter list • Bytes 1 and 2 indicate a configuration change to the Sysplex Timer (ETR). • Byte 3, if non-zero, indicates a change to the local time offset. Possible values are: **0** Sysplex Timer configuration has changed. **1** Local time offset has changed. **2** Leap second offset has changed. **3** Both local time offsets have changed.	EXIT / NO
55	The system resource manager (SRM) has detected a significant MVS image event, which is being signalled. The qualifiers and parameters further define the event.	**ENF55QLF_REAL_SHORTAGE (X'80000000')** Too many fixed frames in storage; issued when IRA400E occurs. **ENF55QLF_REAL_SHORTAGE_RELIEVED (X'40000000')** Pageable storage shortage due to excessive fixed storage relieved; issued when IRA402I occurs. **ENF55QLF_REAL_WARNING (X'20000000')** Pageable storage warning that indicates there are many fixed frames in storage; issued when IRA405I occurs. **ENF55QLF_AUX_CRITICAL_SHORTAGE (X'08000000')** Too many slots allocated in the AUX subsystem. It is a critical shortage and is issued when IRA201E occurs. **ENF55QLF_AUX_SHORTAGE (X'04000000')** Too many slots allocated in the AUX subsystem. It is issued when IRA200E occurs. **ENF55QLF_AUX_SHORTAGE_RELIEVED (X'02000000')** AUX Storage shortage due to excessive slots relieved. It is issued when IRA202I occurs. **ENF55QLF_AUX_WARNING (X'01000000')** AUX Storage usage warning that indicates there are many slots allocated in the AUX subsystem. It is issued when IRA205I occurs. **ENF55QLF_SCM_HIGH_USAGE (X'00040000')** High usage of storage-class memory (SCM). Issued when IRA250I occurs.	Mapped by IRAENF55	EXIT or SRBEXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
55 (cont.)		**ENF55QLF_SCM_HIGH_USAGE_RELIEVED (X'00020000')** High usage of storage-class memory (SCM) relieved. Issued when IRA252I occurs. **ENF55QLF_AFQ_SHORTAGE (X'00008000')** Available frame queue shortage. Not enough frames on the available frame queue. **ENF55QLF_AFQ_SHORTAGE_RELIEVED (X'00004000')** Available frame queue shortage relieved. **ENF55QLF_PREF_SHORTAGE (X'00002000')** Preferred frame queue shortage. Not enough frames on the preferred frame queue. **Note:** This preferred storage shortage indicator is an informational notification for applications that are able to change their storage allocation type. In case of a preferred storage shortage, the application should request non-preferred storage (if possible) instead of preferred storage. No action is taken by SRM to address this shortage. **ENF55QLF_PREF_SHORTAGE_RELIEVED (X'00001000')** Preferred frame queue shortage relieved.		
56	Workload management has changed an attribute of a job.	**WLMENF56_QUAL_RESET** A job was reset using the RESET system command or IWMRESET macro. **WLMENF56_QUAL_ ENCLAVERESET** An enclave has been successfully reset via the IWMERES service.	Mapped by IWMRENF2	EXIT / NO
57	The state of a workload management scheduling environment has been altered.	**WLMENF57_NORMAL_ SCHENV_CHANGE** The state of a scheduling environment has changed due to a F WLM,RESOURCE command or IWMSESET macro. **WLMENF57_RECOVERY_ SCHENV_CHANGE** The state of a scheduling environment has changed due to workload management recovery processing.	Mapped by IWMRENF57	EXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
58	The state of a SYSOUT data set has changed. The state of a SYSOUT data set changes when it is either dynamically allocated using the DALRTCTK text unit, or when the SYSOUT application program interface (SAPI) disposition bit is set. For more information, see the "Listening for Events" section of the "JES Client/Server Print Interface" chapter in *z/OS JES Application Programming*.	**ENF58_Q_PURGE** The data set was purged. **ENF58_Q_SELECT** The data set was selected. **ENF58_Q_DESELECT_ PROCESSED** The data set was processed. **ENF58_Q_DESELECT_ NOT_PROCESSED** The data set is no longer selected, disposition was not updated. **ENF58_Q_DESELECT_ NOT_PROCESSED_HELD** The data set is no longer selected, disposition was not updated, and data set is held. **ENF58_Q_DESELECT_ ERROR** An error resulting in a system level hold occurred. **ENF58_Q_EOD_OK** End of data set notification occurred — successful. **ENF58_Q_EOD_ERROR** End of data set notification occurred — unsuccessful. **ENF58_Q_JOB_CHANGE** A job status change occurred. **ENF58_Q_TOKEN_ CHANGE** The client token has changed. **ENF58_Q_INSTANCE** Addition instance of data set created. **ENF58_Q_GRP_SELECT** Data set group select. **ENF58_Q_GRP_DESELECT** Data set group deselect.	Mapped by IAZENF58	EXIT / YES
60	A TRACE TT command has been accepted.	ENF60_QUAL	Mapped by ITZENF60	EXIT / NO
61	The capacity of the MVS image or CEC has changed.	WLMENF61_CAPACITY_ CHANGE	Mapped by IWMENF61	EXIT / NO
62	A RACF SETROPTS RACLIST command has affected in-storage profiles used for authorization requests in a class designated as SIGNAL=YES or SIGNAL(YES) in the RACF class descriptor table. The class affected is in the parameter list in field IRR_ENFCLASS.	The qualifier (QUAL) has the following format: **BYTE 1** **X'80'** SETROPTS RACLIST has taken place. **BYTE 2** **X'80'** SETROPTS RACLIST REFRESH has taken place. **BYTE 3** **X'80'** SETROPTS NORACLIST has taken place.	Mapped by IRRPENFP in SYS1.MACLIB.	EXIT or SRBEXIT/ NO
63	A permanent error was detected on a HyperSwap capable device.	None	Mapped by IOSDE63R	EXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
64	One of the following events occurred: • The capacity of a storage volume has changed. • The VTOC or INDEX of a direct access volume has been extended or moved to a new location. • The VTOC index of a storage volume has been built. The index indicates a direct access volume has changed from an OS format VTOC (OSVTOC) to an indexed format VTOC (IXVTOC). • The content of the volume has changed due to a full volume copy or restore operation. • DS8K recovery scenario occurs either on primary or secondary PPRC disk subsystem. DS8K signals z/OS on all paths via the Storage Controller Health Message attention status. • One or more devices in the logical subsystem has a PPRC state change.	The qualifier (QUAL parameter) has the following format: **BYTE 1** Type of signal information: **X'01'** Volume event **X'02'** LSS event **BYTE 2** Varies with event **BYTE 3** Always X'00' **BYTE 4** Varies with event The defined QUAL values are: **Qualifier** **Information type** **X'0100*xxx*'** Volume events **X'01000001'** DASD volume capacity changed **X'01000002'** VTOC updated (moved or extended) **X'01000003'** VTOC index is built **X'’010000010'** Full volume copy or restore **X'02xxxxxx'** LSS event **X'02010001'** Storage controller health (LSS) event **X'02020001'** Summary (LSS) event - PPRC state change	Mapped by IECENF64	EXIT / YES

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
65	System REXX event has occurred.	**X'80000000'** The AXR address space has initialized. AXREXX can be invoked. **X'40000000'** The AXR address space has terminated. Subsequent AXREXX invocations will be rejected. **X'20000000'** The AXR address space has reached its threshold of ACTIVE+WAITING AXREXX requests. No more requests will be accepted until the number of requests drops to an acceptable level. **X'10000000'** The number of ACTIVE+WAITING AXREXX requests has dropped to an acceptable level. AXREXX requests are now being accepted. **X'08000000'** The number of ACTIVE+WAITING AXREXX requests is high and is nearing the level where subsequent requests will be rejected. **X'04000000'** The number of extents in the REXXLIB concatenation exceeds the system limit. See *z/OS DFSMS Using Data Sets* for details. If this condition is detected during System REXX initialization, System REXX terminates; otherwise, no new AXREXX requests will be accepted.	None	EXIT/NO
67	One of the following IBM Health Checker for z/OS events has occurred: • IBM Health Checker for z/OS has become available. • IBM Health Checker for z/OS has terminated and is not available.	The defined BITQUAL values are: **Qualifier** **Information type** **X'80000000'** IBM Health Checker for z/OS is available. Field Enf067_BitQual_Available in the HZSZENF mapping macro. **X'40000000'** IBM Health Checker for z/OS has terminated and is not available. Field Enf067_BitQual_NotAvailable in the HZSZENF mapping macro.	Mapped by HZSZENF	EXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
68	One of the following BCPii events has occurred: • A change in BCPii status has occurred. • A hardware communication error has occurred. • A hardware event has occurred.	The defined QUAL values are: **Qualifier** **Information type** **X'01000001'** BCPii is available. **X'01000002'** BCPii is not available. **X'020100*yy*'** A hardware communication error has occurred and CPC events might have been lost. *yy* denotes the type of error: **01** A temporary error, some events might have been lost. **02** A permanent error, no more events are delivered. **03** Communication to the CPC has been established or re-established. Event delivery from this CPC will now commence or re-commence. **X'03*xx*00*yy*'** A hardware event has occurred. *xx* denotes the event source: **01** CPC **02** Image *yy* denotes the event. The defined BITQUAL values are: **Qualifier** **Information type** **X'01nnnnnn'** N/A **X'0201nnnn'** Bytes 1-17 CPC name, padded with hexadecimal zeros **X'0301nnnn'** Bytes 1-17, CPC name, padded with hexadecimal zeros **X'0302nnnn'** Bytes 1-17, CPC name, padded with hexadecimal zeros Bytes 18-24, image name, padded with hexadecimal zeros	Mapped by HWICIASM and HWICIC	EXIT / NO

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
68	One of the following BCPii events has occurred: - A change in BCPii status. - A hardware communication error has occurred. - A hardware event has occurred.	Hardware Event Codes: **Code** **Description** **'01' x** A command response has been received. **'02' x** An object status change has occurred. **'03' x** An object name change has occurred. **'04' x** The activation profile of the object has changed. **'05' x** A new object was created. **'06' x** An object was deleted. **'07' x** An object entered or left an exception state. **'08' x** A Console application has started. **'09' x** A Console application has ended. **'0A' x** An operating system message has been received. **'0B' x** A hardware message has been received. **'0C' x** A hardware message has been deleted. **'0D' x** A capacity change event has been received. **'0E' x** A capacity record change has occurred. **'0F' x** A security event has been logged. **'10' x** An image has entered a disabled wait state. **'11' x** A power change event has been received.	Mapped by HWICIASM and HWICIC	EXIT / NO
70	The state of a job (batch, STC or TSU) owned by JES has changed. The job may have been selected for processing, completed processing, changed phase (including changes to execution phase job class), or been purged.	**ENF70_SELECT** Job was selected. **ENF70_DESELECT** Job was processed. **ENF70_CHANGE** Job queued to new phase of processing. **ENF70_PURGE** Job was purged.	Mapped by IAZENF70	EXIT / YES
71	A RACF command has affected a user's group connections which may affect his or her resource authorization. The user affected is in the parameter list in field IRR_ENF2USER. The group affected is in the parameter list in field IRR_ENF2GROUP. Control flags that are used to provide greater granularity for the listeners are in the parameter list in field IRR_ENF2Flags.	The qualifier (QUAL) has the following format: **BYTE 1** **X'80'** CONNECT command **X'40'** REMOVE command **X'20'** ALTUSER REVOKE command **X'10'** DELUSER command **X'08'** DELGROUP command **BYTES 2 - 4** Reserved	Mapped by IRRPENF2 (See *z/OS Security Server RACF Data Areas*)	EXIT or SRBEXIT / YES

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
72	Volume status information for SMS. The listener user exit routine for event code 72 can run in SRB mode. Event code 72 can generate large numbers of events in short periods of time. The listener user exit routine for event code 72 must be able to handle the volume of events.	The qualifier (QUAL) has the following format: **BYTE 1** Type of signal information: **X'01'** Volume status **BYTE 2** Always X'00' **BYTES 3 - 4** Varies with event. The defined QUAL values are: **Qualifier** **Information type** **X'01000001'** ENF72_OVER_THRESHOLD_AM_Y. Volume over threshold when storage group is defined using AM=Y (Auto Migrate, yes).	Volume Status information: IGDE72VL Mapped by IGDENV72	EXIT or SRBEXIT / YES
73	The SETLOAD xx, IEASYM command has completed successfully. The local system's symbol table has been updated.	None	None	EXIT/ NO
78	The state of a job (batch, STC or TSU) owned by JES2 has changed. The job has completed processing.	ENF78_JOB_NOTIFY	Mapped by IAZENF78	EXIT/YES

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
79	A RACF command has modified a profile such that a user's authorization to the resources it protects may be affected. • The user affected is in the parameter list in field IRR_ENF3_UserID. • The class in which the modified profile belongs is in the parameter list in field IRR_ENF3_ClassName. • The length of the affected profile name is in the parameter list in field IRR_ENF3_ProfName_Length. • The name of the affected profile is in the parameter list in field IRR_ENF3_ProfName. Control flags that are used to provide greater granularity for the listeners are in the parameter list in field IRR_ENF3_Flags. For the PERMIT RACF command processor, there maybe additional information regarding: • The type of Conditional Access, a numerical value that is in the parameter list in field IRR_ENF3_PERMIT_WHEN_Cond. • The Conditional Access List Entry. The length of the Conditional Access Name and the Conditional Access Name itself is in the parameter list in the fields: IRR_ENF3_CACLName_Length IRR_ENF3_CACLName For the RDEFINE and RALTER RACF command processors, there may be additional information in the ADDMEM and DELMEM lists. The number of elements in the list, the length of the list, and the offset to the list are in the parameter list in the fields: IRR_ENF3_ADDMEML_Member# IRR_ENF3_DELMEML_Member# IRR_ENF3_ADDMEML_Length IRR_ENF3_DELMEML_Length IRR_ENF3_ADDMEML_Offset IRR_ENF3_DELMEML_Offset	The qualifier (IRR_ENF3_QualCode) has the following format: **BYTE 1** **X'80'** PERMIT command **X'40'** RDEFINE command **X'20'** RALTER command **X'10'** DELETE command **BYTES 2 - 4** Reserved	Mapped by IRRPENF3 (See *z/OS Security Server RACF Data Areas*)	EXIT or SRBEXIT / YES

Table 21. ENF macro event codes (continued)

Event code	Description	Qualifier	Parameter list passed to the user exit	Exit type / Cross-system capable
80	One of the following z/OS Communication Server events has occurred: • The rpcbind server has initialized. • The rpcbind server is stopping. **Notes:** 1. ENF80_RPC DSECT maps the RPCBIND event. 2. Use the ENF80_RPC_FLAGs to determine if the rpcbind server is initializing or ending. 3. When flag ENF80_RPCINIT is on, RPC applications can register with RPCBIND. 4. When flag ENF80_RPCTERM is on, the rpcbind server is stopping. 5. ENF80_RPC DSECT includes the jobname of the rpcbind server that generated the event.	ENF80_RPC_EVENT	Mapped by EZAENF80	SRBEXIT / NO

ENF sample programs

The following code segments from a sample listening program show how to:

- Set up a program to listen for specific event codes
- Use qualifiers to more specifically define an event
- Pass control to the appropriate listener user exit routine
- Delete a listen request

Following this sample listening program are the listener user exit routines which this program calls.

SMFLSTEN - Sample ENF listener

This module controls the following two sample listener user exit routines:

- SMFLSTAL - Gains control for all qualifier values
- SMFLST00 - Only gains control if qualifier equals ENF37Q00

For each listener user exit routine, this module issues the ENFREQ LISTEN request, waits for the associated processing to complete and then issues the ENFREQ DELETE request.

A program that listens for ENF events should perform the following:

- Initialize Module
  - Perform Entry Linkage
    - Save input registers
    - MODESET to key 0, Supervisor State
  - Get Working Storage
- Set Module Recovery

  When the module ends abnormally, its ESTAE routine attempts a RETRY to ensure that ENFREQ DELETE deletes all of the listen requests.
- Set up for ENF Listen Requests

– Ensure that the listener user exits are in common storage

– Issue an ENFREQ LISTEN request (with or without a qualifier)

Issue listen request without qualifier

The following code segment loads the SMFLSTAL listener user exit routine into common storage and issues the ENFREQ LISTEN request to listen for all event code 37 (SMF accounting) events. Because this ENFREQ request does not use the QUAL and QMASK keywords, the program listens for all code 37 events. For a description of the exit routine, see “SMFLSTAL - Sample ENF listener user exit routine” on page 188. For a description of the variables, see “Module declarations” on page 187.

```
* Load ENF Listen Exit (SMFLSTAL) into common storage and save address
* SMFLALL@ contains the address of the listener user exit routine
         .
         .
         ST    R00,SMFLALL@
*
* Issue LISTEN Request for SMF Event Code (all functions)
         L     R02,SMFLALL@
         ENFREQ ACTION=LISTEN,    -- Function                          +
               CODE=ENFC37,       -- Event Code                        +
               EXIT=(R02),        -- Exit Address                      +
               ESTBNME=THISMOD,   -- Establisher Name                  +
               EXITNME=SMFLSTAL,  -- Exit Name                         +
               DTOKEN=SMFLALLT    -- Returned Token Field
*
* Check the return code from ENFREQ - if not zero issue message
*
```

Issue listen request with QUAL=ENF37Q00

The following code segment sets up and loads into common storage the SMFLST00 listener user exit routine, which gains control only if the qualifier equals ENF37Q00.

Note that the qualifiers are declared in the IFAENF37 mapping macro. The ENFREQ macro specifies QMASK=ALL, which specifies that all four bytes of the qualifier mask are to be used in the qualifier comparison.

See “SMFLST00 - Sample ENF listener user exit routine” on page 189 for a description of this listener user exit routine.

```
* Load ENF Listen Exit (SMFLST00) into common storage and save address
* SMFL00@ contains the address of the listener user exit routine
         .
         .
         ST    R00,SMFL00@
*
* Issue LISTEN Request for SMF Event Code (Qualifier ENF37Q00)
         L     R02,SMFL00@
         ENFREQ ACTION=LISTEN,    -- Function                          +
               CODE=ENFC37,       -- Event Code                        +
               EXIT=(R02),        -- Exit Address                      +
               QUAL=ENF37Q00,     -- Qualifier Value                   +
               QMASK=ALL,         -- Qualifier Mask (Full Word)        +
               ESTBNME=THISMOD,   -- Establisher Name                  +
               EXITNME=SMFLST00,  -- Exit Name                         +
               DTOKEN=SMFL00T     -- Returned Token Field
*
* Check the return code from ENFREQ - if not zero issue message
*
```

Sample ENFREQ DELETE request

After issuing an ENFREQ LISTEN request, the module saves, for later use, the returned DTOKEN and does one of the following:

- Ends Processing
- Waits for completion of processing (using a WTOR or STIMER WAIT request for testing purposes)

When the listening program no longer needs the listener user exit routine, the program deletes the listen request by issuing an ENFREQ DELETE request specifying the DTOKEN that was returned on the ENFREQ LISTEN request.

The following code segment shows the ENFREQ DELETE request for the DTOKEN that was returned on the LISTEN request for the SMFLSTAL exit:

```
*
* Issue DELETE Request for SMF Event Code (all functions)
         ENFREQ ACTION=DELETE,                                         +
               CODE=ENFC37,                                            +
               DTOKEN=SMFLALLT
*
* Check the return code from ENFREQ - if not zero issue message
*
```

The following code segment shows the ENFREQ DELETE request for the DTOKEN that was returned on the LISTEN request for the SMFLST00 exit:

```
* Issue DELETE Request for SMF Event Code (Qualifier ENF37Q00)
         ENFREQ ACTION=DELETE,                                         +
               CODE=ENFC37,                                            +
               DTOKEN=SMFL00T
*
* Check the return code from ENFREQ - if not zero issue message
*
```

The following code segment shows the ENFREQ DELETE request for the DTOKEN that was returned on the LISTEN request for the SMFLST01 exit:

```
* Issue DELETE Request for SMF Event Code (Qualifier ENF37Q01)
         ENFREQ ACTION=DELETE,                                         +
               CODE=ENFC37,                                            +
               DTOKEN=SMFL01T
*
* Check the return code from ENFREQ - if not zero issue message
*
```

The listening program must do the following to complete clean up and end the module:

- Release ESTAE for module
- Free Automatic Storage
- Perform Exit Linkage
  - MODESET back to problem state
  - Return to caller

Module declarations

The following segment defines the variables and control blocks that are required for this program:

```
* - Local variables
DATAAREA DSECT
SMFLALL@ DS    A
SMFL00@  DS    A
```

```
SMFL01@  DS    A
SMFLALLT DS    F
SMFL00T  DS    F
SMFL01T  DS    F
ENFPTR   DS    A
*
* - Local constants
SMFLSTEN CSECT
         DS    0F
ENFC37   EQU   37
THISMOD  DC    CL8'SMFLSTEN'
SMFLSTAL DC    CL8'SMFLSTAL'
SMFLST00 DC    CL8'SMFLST00'
SMFLST01 DC    CL8'SMFLST01'
*
* - External control blocks
         CVT   DSECT=YES
         IEFENFCT
SMFLSTEN CSECT
         IFAENF37
DATAAREA DSECT
         IEFENFPM
LENODATA EQU   *-DATAAREA
*
```

Note that the IFAENF37 macro includes the following declarations:

```
&SYSECT  CSECT                Control Section for Constants
ENF37Q00 DC    X'80000000'    SMF Active
ENF37Q01 DC    X'40000000'    SMF Terminated
```

SMFLSTAL - Sample ENF listener user exit routine

This ENF listener user exit routine processes all the qualifier values for the ENF event code.

By checking the qualifier value passed on the parmlist, this exit routine is able to determine which event occurred and process accordingly.

After checking the qualifier, this exit issues a message to the console to tell the operator which of the following SMF Accounting events occurred:

ENF37C00
'SMF has completed initialization'

ENF37C01
'SMF has terminated'

On entry to this listener user exit routine, register 1 points to the address of the input parmlist (mapped by the macro IFAENF37).

To initialize this exit routine you must do the following:

- Perform Entry Linkage
  - Save input registers
- Get Working Storage
- Get the address of the input parm list and define DSECT mapping

```
*
* Get the input parameter structure and set DSECT basing
         L     E37@,0(R01)             Get Input Parm Structure
         USING ENF37,E37@              Set Parm Structure basing
*
```

After completing initialization, this exit routine determines which event occurred and processes accordingly.

- If the qualifier indicates 'SMF has completed initialization' then issue a message to the console.

```
*
         CLC   ENF37QLF(4),ENF37Q00
         BNE   CHKQ01
         WTO   'SMFLSTAL: SMF HAS COMPLETED INITIALIZATION'
         B     MODTERM
*
```

- If the qualifier indicates 'SMF has terminated' then issue a message to the console.

```
*
CHKQ01   EQU   *
         CLC   ENF37QLF(4),ENF37Q01
         BNE   CHKQ02
         WTO   'SMFLSTAL: SMF HAS TERMINATED'
         B     MODTERM
*
CHKQ02   EQU   *
```

To end cleanly, the exit routine must do the following:

- Free Automatic Storage
- Restore Input Registers
- Perform Exit Linkage
- Return to caller

Module declarations

The following segment defines the variables and control blocks that are required for this program:

```
* - Local variables
DATAAREA DSECT
ENF37@   EQU   4
*
* - External control blocks
SMFLSTAL CSECT
         IFAENF37
*
```

Note that the IFAENF37 mapping macro includes the following declarations:

```
ENF37    DSECT                SMF ENF Parameter List for Inter
*
ENF37QLF DS    CL4 -          Qualifier Code
*
&SYSECT  CSECT                Control Section for Constants
ENF37Q00 DC    X'80000000'    SMF Active
ENF37Q01 DC    X'40000000'    SMF Terminated
```

SMFLST00 - Sample ENF listener user exit routine

This ENF listener user exit routine is set up to process ONLY the qualifier value ENF37Q00 for the ENF Event Code.

This exit routine issues a message to the console to tell the operator that SMF completed initialization.

On entry to this listener user exit routine, register 1 points to the address of the input parmlist (mapped by the macro IFAENF37). Because the specific event is already known in this case, however, this exit routine does not reference the parmlist.

This exit routine performs module initialization by:
- Performing Entry Linkage
- Saving input resisters
- Obtaining Working Storage

After completing initialization, the module issues a message to the console indicating that SMF completed initialization.

```
*
         WTO   'SMFLST00: SMF HAS COMPLETED INITIALIZATION'
*
```

After issuing the WTO message, the listener user exit routine ends cleanly by:
- Restoring Input Registers
- Freeing Automatic Storage
- Performing Exit Linkage
- Returning to caller

Listening for global resource serialization-related system events

GRS will issue ENF signal 51 when resource contention occurs or when an RNL change causes a user job to be suspended or to resume. The following topic provide guidance for using ENF to listen for the occurrence of these events. See Chapter 8, “Listening for system events,” on page 159 for a detailed discussion of the ENFREQ ACTION=LISTEN interface, and “ENF event codes and meanings” on page 168 for a description of the qualifier values defined for the global resource serialization event code.

Monitoring contention changes

There are two types of contention events:
- Where the requester waits for the contention to complete before becoming an owner of the resource.
- Where the requester requested not to wait for the resource to become available. These result from RET=USE and RET=CHNG type ENQs when the resource was unavailable. Listeners can request to get control only for a specific type or for both types depending on the qualifiers that they specify.

To establish an ENF listen exit to be called when resource contention occurs, issue

```
ENFREQ ACTION=LISTEN
```

with one of the qualifier/qmask combinations described in Table 22 on page 191. In the table:
- In qualifier values, the character 'x' indicates an arbitrary value (“don't-care” value).
- An incremental update might represent multiple changes in the state of contention (for instance, multiple new waiters, or multiple changes in ownership).

If you are monitoring contention for global resources, your listen request must specify XSYS=YES even if you are monitoring contention involving only the local system. Contention notifications may originate from another system that is involved in contention for a resource, or even from a system that is not involved in contention at all.

It is possible to receive duplicate notification, that is, two signals that represent the same contention data. For example, this can occur if an ENQ and DEQ happen in rapid succession, thereby causing and relieving contention. This could result in two notifications, both indicating that no contention exists.

Table 22. Qualifier / QMASK combinations for contention data

Qualifier (hexadecimal)	QMASK	Result
None	NONE	All signal 51 events, including both contention and RNL change data. Qualifier value requests all contention data, including waitless contention resulting from RET=USE and RET=CHNG type ENQs.

01*xxxxxx*	BYTE1	All contention-related events, including waitless contention resulting from RET=USE and RET=CHNG type ENQs. All contention-related events. To correspond to a GQ SCAN request, specify the following:
		`SCOPE=(Global,Local)`
		`WAITCNT=1`
		`XSYS=YES`

01*xxxx*01	BYTE1, BYTE4	All contention-related events, including waitless contention resulting from RET=USE and RET=CHNG type ENQs. Incremental updates to contention information for local resources. Corresponds to a GQSCAN request specifying:
		`RESNAME=resource name specified`
		`on the ENQ or DEQ request`
		`SCOPE=LOCAL`

01*xxxx*02	BYTE1, BYTE4	All contention-related events, including waitless contention resulting from RET=USE and RET=CHNG type ENQs. Incremental updates to contention information for global resources. Corresponds to a GQSCAN request specifying:
		`RESNAME=resource name specified`
		`on the ENQ or DEQ request`
		`SCOPE=GLOBAL`
		`XSYS=YES`

01*xxxx*03	BYTE1, BYTE4	Contention-related recovery information, including removal of a system from the sysplex or limitation of contention notification capability.
01*xx*00*xx*	BYTE1, BYTE3	All contention data, excluding waitless contention resulting from RET=USE and RET=CHNG type ENQs.
01*xx*0001	BYTE1, BYTE3, BYTE4	All contention data, excluding waitless contention resulting from RET=USE and RET=CHNG type ENQs.

Table 22. Qualifier / QMASK combinations for contention data (continued)

Qualifier (hexadecimal)	QMASK	Result
01*xx*0002	BYTE1, BYTE3, BYTE4	All contention data, excluding waitless contention resulting from RET=USE and RET=CHNG type ENQs.

If you are monitoring contention changes and are maintaining a database of current contention data, observe the following protocol to obtain a set of initial data:

1. Establish your ENF listen request.
2. On return from ENF, record a timestamp (STCK).
3. Issue GQSCAN SCOPE=LOCAL and/or GLOBAL, WAITCNT=1,XSYS=YES.
4. In your listen exit, process and store any contention notifications accumulated during GQSCAN processing.
5. On return from GQSCAN, apply the stored contention notifications to the GQSCAN output. Discard any notifications timestamped earlier than the recorded timestamp, and apply any notifications with a timestamp equal to or later than the recorded timestamp.
6. Complete initialization and begin normal processing, applying contention notifications as they arrive.

If you are monitoring resource contention, you should define your listen exit to receive and process recovery signals identified by the X'01xxxx03' qualifier. The recovery signals are characterized further by the following flags in the ISGE51CN parameter list:

ENF51C_SYSTEM_FAILURE

If the ENF51C_SYSTEM_FAILURE flag is set, the system identified by the ENF51C_FAILED_SYSTEM field has failed. If the ENF listener is maintaining a database of contention information, it must be reinitialized as described above.

Note: Global resource serialization may not issue this event for all system failures. This flag is only set when global resource serialization cannot determine the effect of the system failure on contention already in existence.

ENF51C_SYSTEM_ERROR

If the ENF51C_SYSTEM_ERROR flag is set, contention information has been lost due to a system error:

- In a global resource serialization ring complex, contention monitors on the system where the failure occurred will receive no further contention signals. Contention information relating to that system will be available via ENF signals on the other systems in the complex. The signal is only sent to listeners on the affected system.
- In a global resource serialization star complex, no further contention information for the system where the error occurred will be available via ENF signals to other systems in the complex. Global resource contention data is not valid on any system. Local resource contention data remains valid on systems other than the one suffering the error. The signal is sent to all systems in the sysplex.

The CVTGRSST field contains the following flags that describe the availability of ENF contention data on each system following the error:

CVTE51GN
When this flag is set, global resource contention data is not available. ENF 51 listeners may receive signals for global resource contention, but the data will be incomplete.

CVTE51LN
When this flag is set, local resource contention data is not available. ENF 51 listeners will not receive signals for local resource contention.

`ENF51C_SYSTEM_ERROR_CLEARED`
If this flag is set, all systems that had suffered the error reported by the ENF51C_SYSTEM_ERROR signal have been partitioned from the sysplex. Global resource contention data is once again valid on all systems. This signal is sent to listeners on all systems in a star complex. (In a ring complex, the other systems were not affected by the original error.)

Listening for RNL change data

To establish an ENF listen exit to be called when a user job is affected by RNL changes, issue

```
ENFREQ ACTION=LISTEN
```

with one of the qualifier/qmask combinations described in Table 23. In the table:

- In qualifier values, the character 'x' indicates an arbitrary value (“don't-care” value).

Table 23. Qualifier / QMASK combinations for RNL change data

Qualifier (hexadecimal)	QMASK	Result
None	N/A	All signal 51 events, including both contention and RNL change data
02xxxxxx	BYTE1	All RNL-related events
02xxxx01	BYTE1,BYTE4	User job suspended due to RNL changes
02xxxx02	BYTE1,BYTE4	User job resumed following RNL changes

Listening for other global resource serialization events

To establish an ENF listen exit to be called global resource serialization mode changes (that is, migrations from a ring to star complex), issue

```
ENFREQ ACTION=LISTEN
```

with one of the qualifier/qmask combinations described in Table 24. In the table:

- In qualifier values, the character 'x' indicates an arbitrary value (“ don't-care” value).

Table 24. Qualifier / QMASK combinations for Global Resource Serialization Mode changes

Qualifier (hexadecimal)	QMASK	Result
03xxxxxx	BYTE1	Any mode change
03xxyyzz	BYTE1,BYTE3 BYTE4	Changes from yy mode to zz mode, where the values for yy and zz are defined in the IHAECVT for the ECVTGMODE field.

Chapter 9. Using a service request block (SRB)

There are many advantages to the use of multiple virtual address spaces. Virtual addressing permits an addressing range that is greater than the central storage capabilities of the system. The use of multiple virtual address spaces provides this virtual addressing capability to each job in the system by assigning each job its own separate virtual address space. The potentially large number of address spaces provides the system with a large virtual addressing capacity.

With multiple virtual address spaces, errors are confined to one address space, except for errors in commonly addressable storage, thus improving system reliability and making error recovery easier. Programs in separate address spaces are protected from each other. Isolating data in its own address space also protects the data.

In a multiple virtual address space environment, sometimes applications need ways to communicate between address spaces. There are two basic methods of inter-address space communication:

- Scheduling a service request block (SRB), an asynchronous process described in this information.
- Using cross-memory services and access registers, synchronous processes that are described in *z/OS MVS Programming: Extended Addressability Guide*.

What is an SRB?

An SRB is a control block that represents a routine that performs a particular function or service in a specified address space. An SRB is similar to a TCB in that it identifies a unit of work to the system. Some characteristics of an SRB are:

- An SRB cannot "own" storage areas. SRB routines can obtain, reference, use, and free storage areas, but the areas must be owned by a TCB.
- An SRB has associated with it such resources as a dispatchable unit access list (DU-AL), a functional recovery routine (FRR) stack, a linkage stack, and a CPU timer value.

The routine that performs the function or service is called the *SRB routine*; initiating the process is called *scheduling an SRB*.

Why would you use an SRB?

A program uses an SRB to initiate a process in another address space or in the same address space. The advantage of scheduling an SRB is that an SRB routine is asynchronous in nature and runs independently of the scheduling program. This advantage makes SRBs very useful in the following situations, where the scheduling program does not need to wait for the SRB routine to finish running:

- To process in parallel

  In a multi-processor environment, the SRB routine, after being scheduled, can:

  - Be dispatched on another processor and can run concurrently with the scheduling program. The scheduling program can continue to do other processing in parallel with the SRB routine.
  - Run with the same cross-memory environment as the scheduling program and with a DU-AL that is a copy of the scheduling program's DU-AL as it

exists when the scheduling program issues the SCHEDULE or IEAMSCHD macro, with the exception of any subspace entries in the scheduling program's DU-AL. The system does not copy subspace entries. In this way, the SRB routine and the scheduling program can access the same address and data spaces.

- To avoid serializing

  Because the SRB represents a separate unit of work, the unit of work that schedules the SRB routine is not serialized or delayed while the SRB routine completes its function. The following types of delays can usually be avoided:
  - Page fault resolution
  - Address space swap-ins
  - Lock suspensions - wait time
- To account for resources

  Because the SRB represents a separate unit of work, the processor time spent accomplishing that work can be charged to:
  - The address space in which the SRB is running
  - The enclave in which the SRB is running
  - The client address space in which the SRB is running
- To make changes of state

  In some instances, a program might be running in some state that prohibits it from performing certain functions. The following are examples of when that program might schedule an SRB routine:
  - A program in a disabled state cannot request a suspend-type lock. The program can avoid these restrictions by scheduling an SRB to complete the function.
  - A program running in cross-memory mode requires functions that only can be performed when in non-cross-memory mode. The program can schedule an SRB that places the SRB routine in non-cross-memory mode.
- To raise the priority of a process

  Because the SRB represents a separate unit of work, the SRB has its own dispatching priority. It can run at a priority higher than that of any address space or at the priority of the address space in which it is scheduled.

  It can also run at a preemptable priority, allowing work at an equal or higher priority to have access to the processor.

Although a major advantage of SRBs is their asynchronous nature, you can use WAIT and POST or SUSPEND and RESUME to synchronize the processing of the SRB routine with the scheduling program.

Scheduling and managing SRBs

This information describes how you can schedule an SRB and purge that SRB when you no longer need it. It describes how you use the macros that initiate the running of an SRB routine and provide its recovery. It also describes how you can serialize the processing of an SRB routine and how to transfer control from an SRB to a TCB.

The information includes descriptions of the following macros:

- To schedule an SRB, use either the IEAMSCHD or the SCHEDULE macro. IBM recommends that you use the IEAMSCHD macro. The program that issues either the IEAMSCHD or SCHEDULE macro is called the scheduling program. The scheduling program can schedule an SRB at different priorities.

- The PURGEDQ macro allows for cleanup of SRB activity.
- The SUSPEND macro suspends the running of an SRB routine until an event occurs.
- The RESUME macro causes the SRB routine to resume running.
- The TCTL macro transfers control from an SRB routine to a task.

The following macros are related to scheduling and managing SRBs:

- The IWMECREA macro allows a program to create an enclave. An enclave is a way to associate related SRBs to be managed and dispatched as a group. A program can schedule an SRB to run in an enclave with the IEAMSCHD macro.
- The IWMEDELE macro allows a program to delete an enclave.

For information about how to use the IWMECREA and IWMEDELE macros, see *z/OS MVS Programming: Workload Management Services*.

Before you can code the macros, you need to understand the possible cross-memory environments that the SRB routine can begin running in. Also, you need to be aware of some characteristics of SRB routines and some restrictions that the system places on them.

Specifying the addressing environment of the SRB routine

You can choose the cross-memory environment in which the SRB routine will run and the SRB routine's ability to access address spaces and data spaces. For the IEAMSCHD macro, you specify the environment with the ENV parameter. For the SCHEDULE macro, you specify the environment with the MODE parameter.

Using the ENV parameter on IEAMSCHD

On the ENV parameter on IEAMSCHD, you can specify the following addressing environments for the SRB routine:

HOME
Specifies that the SRB routine is to receive control in the current home address space, and with PASID=SASID=HASID.

PRIMARY
Specifies that the SRB routine is to receive control in the current primary address space, with PASID=SASID=HASID.

FULLXM
Specifies that the SRB receives control with the scheduling program's current cross-memory environment. This means that when the SRB routine begins to run, it has the primary, home, and secondary address space of the scheduling program's at the time of the invocation of the IEAMSCHD macro. In addition, the SRB routine's DU-AL is a copy of the scheduling program's DU-AL as it exists when the scheduling program issues the IEAMSCHD macro, with the exception of any subspace entries in the scheduling program's DU-AL. The system does not copy subspace entries.

STOKEN
Specifies that the SRB routine is to receive control in the address space specified by the target STOKEN and with PASID=SASID=HASID.

Using the MODE parameter on SCHEDULE

On the MODE parameter on SCHEDULE, you can specify the following addressing environments for the SRB routine:

NONXM

Specifies that the SRB routine receives control in non-cross-memory mode, with its primary, secondary, and home address spaces equal to the contents of SRBASCB. In addition, the SRB routine has access to an empty DU-AL.

FULLXM

Specifies that the SRB receives control with the scheduling program's current cross-memory environment. This means that when the SRB routine begins to run, it has the primary, home, and secondary address space of the scheduling program's at the time of the invocation of the SCHEDULE macro. In addition, the SRB routine's DU-AL is a copy of the scheduling program's DU-AL as it exists when the scheduling program issues the SCHEDULE macro, with the exception of any subspace entries in the scheduling program's DU-AL. The system does not copy subspace entries.

With MODE=NONXM, the SRB is required only for initial dispatch. The scheduling routine can free or reuse the SRB after the SRB routine is dispatched. With MODE=FULLXM, the SRB is needed only for the SCHEDULE invocation. The scheduling program can free the SRB immediately after issuing the SCHEDULE macro.

Characteristics and restrictions of SRB routines

SRB routines run enabled and can be interrupted by an asynchronous interruption. They can be scheduled as preemptable SRBs, allowing higher priority work access to the processor. SRB routines might lose control because of synchronous events that cause suspension of the program in control, such as page faults and unconditional requests for suspend-type locks. In this case, full status of the process is saved and other work is dispatched; the SRB routine is redispatched when the situation is resolved.

An enabled SRB routine can take page faults.

- If the routine does not hold any locks when the page fault occurs, the system suspends the SRB routine, which allows the system to dispatch other work on the active processor. The system redispatches the SRB routine after it resolves the page fault.
- If the routine holds a suspend type lock (such as a local, CML, or CMS lock) when a page fault occurs, the suspended SRB routine continues to hold those locks. The system suspends other work units that require the lock held by the suspended SRB routine until the system redispatches the SRB and the SRB routine explicitly releases those locks.

You can use the SUSPEND macro to suspend the execution of an SRB routine until an event occurs. The RESUME macro then causes the SRB routine to resume execution. “Serializing SRB processing” on page 208 describes how you use the SUSPEND and RESUME macros.

If an SRB routine requires an Integrated Cryptographic Feature (ICRF) to encrypt or decrypt data, use the following guidelines:

- Use the FEATURE=CRYPTO parameter if the routine's only purpose is to encrypt or decrypt data.

- Do not use the FEATURE=CRYPTO parameter if the routine has many purposes. In this case, let the system schedule the routine to run on the appropriate processors as the routine's requirements change.

Implications of running in SRB mode

The SRB routine runs in the operating mode known as SRB mode. Code in SRB mode:

- Cannot leave supervisor state. The scheduling program can specify that the SRB routine be dispatched with a LOCAL lock held (LLOCK=YES on the IEAMSCHD or SCHEDULE macro) or have a recovery routine established for the SRB routine, or both.
- Can request any lock through the SETLOCK macro.
- Cannot issue SVCs except ABEND. This limitation means that a program in SRB mode cannot issue some of the system macros and data management macros such as OPEN and CLOSE. Macro descriptions in one of the following books tell whether you can use the macros in SRB mode; if a description does not give this information, you can assume that the macro does not support SRB-mode users.
  - *z/OS MVS Programming: Assembler Services Reference ABE-HSP*
  - *z/OS MVS Programming: Assembler Services Reference IAR-XCT*
  - *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
  - *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*
  - *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
  - *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.
- Must return control to the address supplied in register 14, in supervisor state with no locks held, except the CPU lock. (If LLOCK=YES was specified on the IEAMSCHD or SCHEDULE macro, the routine must release the LOCAL lock.)
- Can create, use, and delete data spaces, but cannot own a data space. For information about data space ownership as it relates to an SRB, see *z/OS MVS Programming: Extended Addressability Guide*. that contains an example of an SRB creating and using a data space, without owning one.
- Can run in access register (AR) address space control (ASC) mode.
- Can issue a PC instruction and schedule an SRB.
- Can resume a suspended task or SRB.
- Can suspend itself.
- Should not be a long-running program. An SRB routine is generally not preempted by I/O interruptions once it is dispatched. An SRB, however, can be scheduled as a preemptable SRB, allowing higher priority work access to the processor.
- Can use an ICRF to encrypt and decrypt data.

If you specify a FULLXM environment and FRR=YES to schedule the SRB, the specified FRR has the characteristics set through the SETFRR macro with the MODE=FULLXM parameter. If you specify any other environment, and FRR=YES, the FRR has the characteristics set through the SETFRR macro with the MODE=HOME parameter.

When you specify FULLXM, a DU-AL with more than 256 entries is not available to the scheduled SRB routine until the SRB routine is dispatched. If an error occurs before the SRB routine is dispatched, the DU-AL might not be available to the SRB routine's FRR.

Environment of the SRB routine at entry

At entry, an SRB routine is in supervisor state, primary ASC mode, enabled and unlocked unless you specified LLOCK=YES, in which case the local lock is held. The general purpose registers contain the following:

Register	**Contents**
0	SRB address if SCHEDULE macro is used; zero if IEAMSCHD is used.
1	Address of an area for passing user parameters from the scheduling program to the SRB routine. (The user parameters are those specified on the PARM parameter in IEAMSCHD or on the SRBPARM parameter in SCHEDULE.)
2	If FRR=YES, 24-bit address of FRR parameter area; otherwise used as a work area by the system
14	Return address
15	Entry point address

Other general purpose registers and all access registers are used as work areas by the system.

Scheduling an SRB (IEAMSCHD or SCHEDULE macro)

You can use either IEAMSCHD or SCHEDULE to schedule an SRB routine. IBM recommends that you use IEAMSCHD. The following table provides a comparison of the two macros:

Table 25. Comparing IEAMSCHD to SCHEDULE

IEAMSCHD	SCHEDULE
IEAMSCHD obtains storage for the SRB for the caller, and frees the storage when SRB processing is complete.	Caller must obtain storage for the SRB.
When the SRB routine gets control, Register 0 contains zeros.	When the SRB routine gets control, Register 0 contains the address of the SRB.
RMTR and FRR are optional.	RMTR and FRR required to free storage at termination of SRB and when an error occurs.
Schedules an SRB at any of the following priorities: local, global, current, preemptable, client, and enclave.	Schedules an SRB at either global or local priority.
IEAMSCHD initializes SRB fields in IHASRB for caller.	Caller must initialize SRB fields in IHASRB.

Scheduling an SRB using IEAMSCHD

To schedule an SRB using IEAMSCHD, follow these procedures:

1. Determine the addressing environment of the SRB, and specify it using the ENV parameter.
2. Determine the priority at which you want to schedule the SRB.

   There are two kinds of priorities: major and minor. The major priority is the dispatching priority. The dispatching priority is dynamically defined based on the goal defined in the service policy. A minor priority is equivalent to a task's dispatching priority. You can specify a minor priority for SRB routines so that they are dispatched before, with, or after tasks in the address space. You can assign a minor priority only to preemptable, client, or enclave priority SRBs.

The SRB may be scheduled for execution in any address space at any of the following major dispatching priorities:

Client
The priority of the address space named by the STOKEN. SRBs scheduled at this priority are preemptable. For more information about using SRBs with a client priority, see *z/OS MVS Programming: Workload Management Services*.

Current
The priority of the scheduling program's current home address space.

Enclave
The priority defined for the work running in the enclave. SRBs scheduled at this priority are preemptable.

Global
The priority of the highest priority work in the system.

Local
The priority of the address space into which the SRB is scheduled. The SRB is not preemptable, and its priority is higher than any preemptable work running in the address space.

Preemptable
The priority of the target home address space.

Preemptable SRBs may also be scheduled with a minor priority.

A minor priority of X'00' is the lowest and X'FF' is the highest. SRB routines with higher minor priority are dispatched before preemptable-class SRB routines with lower minor priority and before tasks in the same address space.

3. Use the FLAGS parameter of IEAMSCHD if you need to differentiate between one or more of these cases that could result from issuing IEAMSCHD:
   - SRB was definitely not queued for execution.
   - SRB might have been queued for execution, but the system cannot know with certainty (this would be a very rare circumstance).
   - SRB was definitely queued for execution.
4. Issue IEAMSCHD with the appropriate parameters.

Scheduling an SRB using SCHEDULE

To schedule an SRB using SCHEDULE, follow these procedures:

1. Allocate storage for the SRB from fixed, commonly addressable storage, with any storage key (0-7).
2. Determine whether or not the SRB routine requires either the same cross-memory environment as the scheduling program or a copy of the scheduling program's DU-AL. If the SRB routine needs either of these, specify MODE=FULLXM on the SCHEDULE macro.

   If the SRB routine does not need the same cross-memory environment as the scheduling program or a copy of the scheduling program's DU-AL:
   - Specify MODE=NONXM on the SCHEDULE macro.
   - Specify the address space in which the SRB routine is to run.
3. Assign the priority of the SRB

Through the SCHEDULE macro, a program schedules either a global SRB (through SCOPE=GLOBAL) or a local SRB (through SCOPE=LOCAL), depending on the priority at which you want the system to dispatch the SRB routine. The system gives a global SRB a priority that is above that of any task in any address space. The system gives a local SRB a priority equal to that of the address space in which the SRB routine will run, but higher than that of any task within that address space.

The global SRBs that user programs dispatch compete with the global SRBs that the system dispatches. Therefore, IBM recommends that you specify SCOPE=LOCAL (the default).

4. Initialize fields in the SRB that identify:
   - The SRB routine
   - A parameter list, if required
   - Additional information for recovery and control

   "Initializing the SRB" provides additional information on obtaining and initializing the SRB.
5. Issue the SCHEDULE macro with the appropriate parameters.

Initializing the SRB

Before a program schedules an SRB, it must obtain 44 bytes of storage for the SRB and initialize its fields. Either the scheduling program or the SRB must free this storage when the SRB is no longer needed.

The SRB can be reused or freed. If you scheduled the SRB with MODE=FULLXM, you can free or reuse the SRB immediately after it has been scheduled. If the scheduling program passed a parameter list to the SRB routine, you must free or reuse only the SRB. The parameter list must remain available to the SRB routine until it has completed processing. If you scheduled the SRB with MODE=NONXM, the scheduling program must provide the serialization to ensure that it doesn't reuse, change, or free the SRB before the SRB routine begins to run.

The IHASRB macro maps the structure of an SRB; include IHASRB in your program. To see the format of the SRB, see the SRB data area see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Use the following information to help you initialize the SRB. First, set the SRB to zeroes, then initialize the fields. The values of all fields not mentioned in this information should be zero.

SRBASCB

Contains the address of the ASCB of the address space in which the SRB routine will run, if the SRB was scheduled without duplicating the scheduling program's cross-memory environment (MODE=NONXM). To make sure that you identify the address space correctly, use the STOKEN parameter on the SCHEDULE macro. The system makes sure that the address space that SRBASCB points to is active and the same as the one that the STOKEN parameter identifies.

Note: If the SRB was scheduled with cross-memory services (MODE=FULLXM), this field does not contain useable information.

SRBPKF

Indicates, in the four high-order bits, the PSW key of the SRB routine. The four low-order bits must be zero.

SRBEP
Specifies the address of the entry point of the SRB routine. If the SRB routine is to run in 31-bit addressing mode, set the high-order bit in the field to 1; if the routine is to run in 24-bit addressing mode, set the high-order bit to 0.

SRBPARM
Contains the address of a user parameter area. Through this field, the scheduling program passes information to the SRB routine.

SRBRMTR
Contains the address of a resource manager termination routine (RMTR). This routine is responsible for cleaning up an SRB that has been scheduled but not yet dispatched. The RMTR is required and must be in common storage. SRBRMTR must contain a valid nonzero address. If the RMTR is to run in 31-bit addressing mode, set the high-order bit in the field to 1. If the RMTR is to receive control with the local lock, set bit 31 in the field to 1 (SRBRMTLL). For information about the RMTR, see "The resource manager termination routine (RMTR)" on page 205.

SRBPTCB
Contains the address of a TCB that is associated with the SRB routine. The system uses this address in two ways:

- If the SRB routine abnormally ends and its FRR does not exist or does not retry, the task is scheduled for abnormal termination.
- If the specified task terminates, the system purges the SRB and gives the RMTR control.

  If this SRB is not related to any task, or purging is not necessary, specify a zero value.

SRBPASID
Contains the ASID of the address space associated with the SRB routine. If you specified a nonzero value in SRBPTCB, you must specify a value for SRBPASID; the value must contain the ASID of the address space containing that TCB. Otherwise, this field can be zeroes. If the specified address space terminates, the system purges the specified SRB and gives control to the RMTR. SRBPASID does not need to represent the same address space as SRBASCB.

SRBFRRA
Contains the address of an FRR that receives control if the SRB routine abnormally ends. If the FRR is to run in 31-bit addressing mode, set the high-order bit in the field to 1; if the routine is to run in 24-bit addressing mode, set the high-order bit to 0. SRBFRRA is needed only when FRR=YES.

Purging an SRB (PURGEDQ macro)

Because an SRB routine is dispatched after the program actually issues either the IEAMSCHD or the SCHEDULE macro, the conditions that existed in the system at the time the macro was issued might have changed by the time the SRB routine begins to run. If, in this time interval, the environment that the SRB routine needs to run successfully has been changed, the results are unpredictable.

An example of a changed environment is when a task or address space terminates, leaving outstanding requests for the task or address space. The system issues PURGEDQs at task and address space termination.

Note: If you used IEAMSCHD, the system filled in the fields of IHASRB. If you used SCHEDULE, the caller filled in the fields of IHASRB.

- For task termination, the system purges any SRBs associated with the task (SRBPTCB) and its address space (SRBPASID).
- For address space termination, the system purges any SRBs scheduled to the address space (SRBASCB, with matching SRBPASID.)

If there are any other conditions for which your SRBs should be purged, issue the PURGEDQ macro. A program, such as an ESTAE routine or a resource manager, might use the PURGEDQ macro to:

- Dequeue SRBs that are scheduled, but not yet dispatched
- Allow processing for previously scheduled SRBs to complete
- Purge each dequeued SRB.
- Purge a preemptable SRB at any point in time.
- Purge a non-preemptable SRB (that voluntarily gave up control by doing a Pause or a SUSPEND with Token) prior to being released or resumed.

Identifying the SRB to be purged

The program must tell PURGEDQ which SRBs are to be purged. Input to PURGEDQ is as follows:

- The address of the RMTR (RMTR parameter, required).
- The address space identifier (corresponding to SRBASCB) of the address space in which the SRB is scheduled to be dispatched (ASID parameter, optional).
- The address space and the TCB associated with the SRB that the system is to purge (ASIDTCB parameter, optional).

The RMTR parameter specifies the address of the RMTR. The RMTR cleans up an SRB that has been scheduled, but not yet dispatched. The system purges only those SRBs whose SRBRMTR field contains the address of the RMTR, as specified on the PURGEDQ macro. When comparing the RMTR parameter to the SRBRMTR field, SRBRMTLL is treated as zero.

The ASID parameter specifies the address of a halfword containing an address space identifier. PURGEDQ searches for SRBs scheduled to be dispatched into the address space specified by this parameter. If you specify the primary address space by coding its address or by omitting the ASID parameter, you need to know what PURGEDQ does with the three kinds of SRBs: active, nondispatched, or suspended:

- It waits for active SRB routines to complete processing.
- It dequeues all nondispatched SRBs. After all of the SRBs have been dequeued or completed, the RMTR specified in the SRB is given control to perform the required cleanup for each dequeued SRB.
- It does not pass control to an RMTR of a suspended SRB. Instead, the system abnormally terminates those SRB routines and waits for the termination to complete.

If you specify an address space other than the primary address space, PURGEDQ will try to purge SRBs that have not been dispatched. PURGEDQ does not guarantee that all SRBs matching the purge parameters will be purged. For example, PURGEDQ will not purge suspended SRBs, nor will PURGEDQ wait for active or suspended SRB routines to complete processing.

When you specify an address space other than the primary, you should use an RMTR to inform the issuer of PURGEDQ whether a particular SRB has been purged; otherwise, problems might result. For example, consider the following sequence of events:

- Program-A running in ASID-Current schedules SRB-A to run in ASID-Target.
- Program-A running in ASID-Current issues the PURGEDQ macro with the ASID parameter to purge SRBs in ASID-Target.
- Program-A running in ASID-Current issues a FREEMAIN for SRB-A.

The preceding sequence of events could result in SRB-A's storage being freed while it is still waiting to be dispatched, because the PURGEDQ function, when the address space specified is other than the primary address space, does not guarantee that all SRBs in ASID-target will be found and purged. This processing might cause spin loops, ABENDs in lock manager, and other problems. Before issuing a FREEMAIN for an SRB, PURGEDQ users who specify an address space other than primary must construct their own communication mechanism to be sure that an SRB has indeed been purged. One way to be sure than an SRB has been purged is to use an RMTR. When the PURGEDQ successfully finds and purges an SRB, the RMTR associated with that SRB is called. The RMTR can free the SRB, or the RMTR can use the SRB parameter area to indicate to another program that the SRB can be safely freed. This prevents SRBs that are missed by PURGEDQ from being freed while they are still waiting to be dispatched.

The ASIDTCB parameter specifies the address of a doubleword that describes the TCB for which SRBs are to be purged. Through this parameter, you can purge the SRBs associated with a specific task. If you omit the parameter, the system purges SRBs associated with the current task in the primary address space.

The resource manager termination routine (RMTR)

If the system has purged the SRB from the dispatching queue before the SRB routine can run, PURGEDQ calls the RMTR associated with the SRB. The primary purpose of the RMTR is to clean up the SRB. The routine can either free the SRB storage by invoking the FREEMAIN or the STORAGE RELEASE macro or mark the SRB so that it can be reused. The choice depends on how your application manages its SRBs.

The RMTR must be commonly addressable from all address spaces and must remain in supervisor state. One RMTR can provide recovery for more than one SRB. However, then you must be more careful when you tell the PURGEDQ macro which SRB (or SRBs) to purge.

At entry, the RMTR is enabled, in task mode, in supervisor state, with PSW key 0. If requested either by bit 31 of RMTRADDR being on for IEAMSCHD or by SRBRMTLL being on, the RMTR will receive control with the local lock held. Otherwise, no locks will be held. Entry register contents are as follows:

Register
: **Contents**

0
: Contents of register 0 of the caller of PURGEDQ at the time the PURGEDQ macro was issued. This register allows the caller of PURGEDQ to pass information to the RMTR. This information can be a pointer or data. Register 0 contains zero if the SRB was purged by the system during address space or task termination.

1
: For IEAMSCHED, 0. For SCHEDULE, the address of the dequeued SRB.

2 For IEAMSCHD, this is the contents specified on the PARM keyword for the SRB. For SCHEDULE, this is the contents specified on SRBPARM for the SRB.

14 Return address to PURGEDQ.

15 Entry point of RMTR.

The RMTR must return control using a BR 14, enabled, in supervisor state with PSW key 0. If the local lock was held on entry, the RMTR is allowed to release the local lock before returning, but is not required to do so. No other locks may be held upon return. If the local lock was not held on entry, the RMTR must return with no locks held. The RMTR is allowed to acquire locks, release locks, issue SVCs and use input registers without first saving the contents during its processing.

Scenario of scheduling and purging an SRB

The best way to describe the activities related to the scheduling of an SRB is through examples.

Example 1

This example shows a scenario of scheduling an SRB using IEAMSCHD and purging the SRB using PURGEDQ. The example schedules an SRB routine:

- To an enclave (identified by ENCLAVE_TOKEN) at the lowest priority in the enclave.
- To receive control in the current home address space with an FRR established and holding the LOCAL lock of the current home address space.
- To run with PSW key 2 in the current home address space
- To have a resource manager termination routine whose entry point address is in RMTR_ADDR.

The scheduling routine's current task recovery is to receive control should the SRB routine's recovery percolate. The SRB routine should be purged if the current task terminates. The ENCLAVE_TOKEN and PURGE_STOKEN were previously initialized.

```
*
SCHED_SRB_RTN  EQU *
               USING  PSA,0          Base Prefixed Save Area
*
               IEAMSCHD  EPADDR=EP_ADDR,FRRADDR=FRR_ADDR,KEYVALUE=
               PSW_KEY_2,PRIORITY=ENCLAVE,ENCLAVETOKEN=ENCLAVE_TOKEN,
               MINORPRIORITY=MINOR_PRIORITY,RMTRADDR=RMTR_ADDR,
               PURGESTOKEN=PURGE_STOKEN,PTCBADDR=PSATOLD,LLOCK=YES,
               ENV=HOME

*
    .
    .
    .
ENCLAVE_TOKEN  DS    D               Enclave Token returned by IWMECREA
PURGE_STOKEN   DS    D               Purge-STOKEN
EP_ADDR        DC    A(SRB_ROUTINE)  Address of Entry Point for SRB
FRR_ADDR       DC    A(FRR_ROUTINE)  Address of FRR Routine
RMTR_ADDR      DC    A(RMTR_ROUTINE) Address of RMTR Entry Point
MINOR_PRIORITY DC    X'00'           Lowest Priority in Enclave
PSW_KEY_2      DC    X'20'           PSW Key 2
```

Example 2

This example shows a scenario of scheduling an SRB using SCHEDULE and purging the SRB using PURGEDQ. Figure 33 shows SRB storage and five pieces of code:

- The scheduling program, PGM1, runs in AS1 and needs the service of a routine in another address space.
- The SRB routine, SRBRTN, runs in address space AS2.
- A recovery routine, RMTR1, in AS1 performs cleanup operations in case AS2 terminates before SRBRTN can run.
- A recovery routine, FRR1, in AS1 receives control if PGM1 abnormally ends.
- A recovery routine, FRR2, in AS2 receives control if SRBRTN abnormally ends.

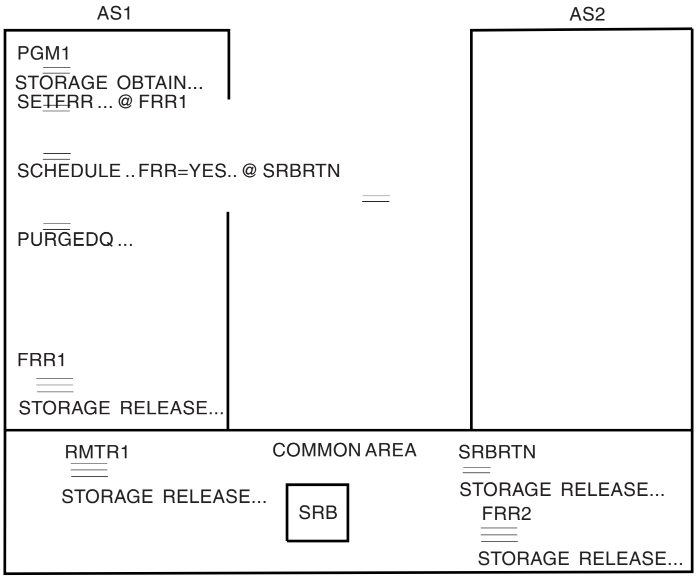


Figure 33. Example of Scheduling an SRB

Note that SRBRTN is not required to be in the common area; it could be in the private area of AS2.

Actions of PGM1:

- STORAGE OBTAIN obtains storage for the SRB from subpool 245.
- PGM1 initializes the SRB. Contents of some fields are:
  - SRBASCB - the ASCB of AS2
  - SRBPASID - the ASID of AS1
  - SRBPTCB - the address of PGM1's TCB
  - SRBEP - the address of SRBRTN
  - SRBRMTR - the address of RMTR1
  - SRBFRRA - the address of FRR2
- SETFRR sets up recovery through FRR1 for PGM1.

- SCHEDULE schedules an SRB. FRR=YES indicates that the scheduling program has set up recovery (through FRR2) for SRBRTN, as defined by SRBFRRA. When the SRB routine gets control, it should fill in the FRR parameter area with the information required by its FRR.

  If the FRR is entered before the SRB routine runs (which would result if, for example, the SRB has affinity but no processors online fit the affinity), then the system fills in the FRR parameter area as follows:
  - Word 1 - Contents of SRBPARM
  - Word 2 - Address of SRB
- PURGEDQ purges the SRBs that were scheduled to be dispatched and that were associated with the TCB.

Actions of SRBRTN:
- Initialize the FRR parameter area
- STORAGE RELEASE frees the SRB storage.

Actions of RMTR1:
- STORAGE RELEASE frees the SRB storage.

Actions of FRR1 that covers PGM1:
- STORAGE RELEASE frees the SRB storage (if it has not already been scheduled).

Actions of FRR2 that covers SRBRTN:
- STORAGE RELEASE frees the SRB storage.
- If FRR2 percolates, the system abnormally ends PGM1 (because SRBPTCB contains the address of PGM1's TCB).

Serializing SRB processing

To serialize an SRB, use the SUSPEND and RESUME macros. The SUSPEND macro identifies the SRB that is to be suspended. In response to the SUSPEND macro, the system provides a token that identifies the SRB and invokes a user-provided exit routine. The exit routine must save the token and then decide whether to suspend the SRB or to allow it to continue running. The RESUME macro enables a program to request that the system resume execution of a suspended SRB or purge the SRB.

Suspending an SRB until an event completes (SUSPEND macro)

The SUSPEND macro enables an SRB routine to suspend execution until an event occurs that requires the routine to resume execution.

To issue SUSPEND from an SRB routine, code both the SPTOKEN parameter and the EXIT parameter. SPTOKEN defines the address of an 8-byte location where the system returns a token called the suspend token. This token uniquely identifies the suspended SRB and is subsequently used by the suspend exit routine and the RESUME macro. The EXIT parameter defines the address of the user-provided suspend exit routine. This exit routine stores the suspend token and decides whether to suspend the SRB or allow it to continue running. The SRB routine can pass a parameter list to the exit routine; the EXITPARM parameter on the SUSPEND macro points to this parameter list. For information on how to write this exit, see “Suspend exit routine” on page 209.

Optionally, you can also code the RSCODE parameter. RSCODE provides the address of a fullword, called the resume code. The resume code provides a means for the exit routine or the program that subsequently resumes the SRB to communicate with your program. The exit routine and the program that issues the RESUME macro have the option of storing a fullword value in the resume code. The meaning of the value is a decision that must be made among the macro callers and the exit designer.

After the SUSPEND macro is issued, one of the following actions occur depending on the suspend exit's decision:

- The SRB routine continues to run, in which case, control immediately returns to the instruction following the SUSPEND macro.
- The SRB is suspended. A suspended SRB remains suspended until a program subsequently issues the RESUME macro to resume SRB execution or to purge the SRB from the system. A resumed SRB resumes execution at the instruction following the SUSPEND macro. A purged SRB never regains control.

Suspend exit routine

The suspend exit routine is required. It is responsible for saving the suspend token. Additionally, it makes the decision whether to suspend an SRB or allow it to continue running. The system invokes the exit routine when an SRB routine issues the SUSPEND macro and identifies the exit on the EXIT parameter. The exit routine must be addressable in the primary address space of the program that issued the SUSPEND macro. Input to the exit routine is the address of the suspend token. The exit routine must store the suspend token and not alter it.

The SRB routine can pass parameters to the exit routine; the EXITPARM parameter on SUSPEND points to the address of the parameter list. The exit routine and the SRB routine determine the format of the list.

Environment on Entry: On entry to the exit routine, the following environment exists. This same environment must exist when the exit routine returns control to its caller.

Environmental factor	**Requirement**
Authorization:	Supervisor state and PSW key 0
Dispatchable unit mode:	SRB (The same SRB that represents the issuer of SUSPEND.)
Cross memory mode:	NewHASN = oldHASN, newPASN = oldPASN, newSASN = oldPASN
AMODE:	31-bit
ASC mode:	Same as the program that issued SUSPEND
Interrupt status:	Enabled for I/O and external interrupts
Locks:	If the calling program holds the local or CML lock, the exit routine holds the lock when invoked.

Registers on Entry: On entry to the exit routine, the general purpose registers contain:

Register	**Contents**
0	Address of the exit routine parameters specified by the EXITPARM parameter of SUSPEND
1	Address of the suspend token
2 - 13	Used as work registers by the system

14 Return address

15 Entry point address

The access registers contain:

Register
Contents

0 ALET that qualifies the address of the exit routine parameters

1 ALET of the suspend token

2 - 15 Used as work registers by the system

To suspend the SRB, the exit routine must place a return code of 0 into register 15. If the caller of the SUSPEND macro holds a local or CML lock, SUSPEND releases the lock when the exit routine passes control back to it and passes a return code of 0. To subsequently resume the SRB, a program will need to use the suspend token. Therefore, the exit routine must store the suspend token in a location that will be addressable by any program that resumes the SRB. For example, the exit routine might store the token in common storage, then set a flag to indicate that the token is stored and the exit has finished processing. On exit, the access registers and general purpose registers 0 - 13 can contain any values placed there by the exit routine. To return to the caller, the exit routine must issue a BR 14 instruction.

To allow the SRB to continue running, the exit must place a return code of 4 into register 15. Optionally, the exit routine can also place a fullword resume code into register 0. The system returns the resume code to the resumed SRB routine. On exit, the access registers and general purpose registers 1 - 13 can contain any values the exit routine places there. To return to the caller, the exit routine must issue a BR 14 instruction.

Resuming or purging a suspended SRB (RESUME macro)

The RESUME macro enables a program to resume execution of a previously suspended SRB or to purge the SRB. Do not attempt to purge or resume a suspended SRB until after the suspend exit receives control.

To resume or purge an SRB, code the SPTOKEN parameter on the RESUME macro. SPTOKEN defines the address of an 8-byte location that contains the suspend token that identifies the SRB. This must be the same suspend token the system provided when the SUSPEND macro was issued to suspend the SRB.

To resume a suspended SRB, in addition to coding SPTOKEN you can code PURGE=NO or omit the PURGE parameter. To communicate with the resumed SRB routine, code the RSCODE parameter. RSCODE defines a fullword (the resume code) where you can place a value that the system will return to the resumed SRB routine. If you omit RSCODE, the system returns a resume code of zero to the resumed SRB routine.

To purge a suspended SRB, in addition to coding SPTOKEN, you must code PURGE=YES. A purged SRB never regains control.

Scenario of suspending and resuming an SRB

The best way to describe the activities related to the suspending and resuming of an SRB is through an example. Figure 34 on page 211 shows the SRB routine SRBRTN that runs in address space AS2. PGM1 in AS1 manages a resource that

SRBRTN needs.

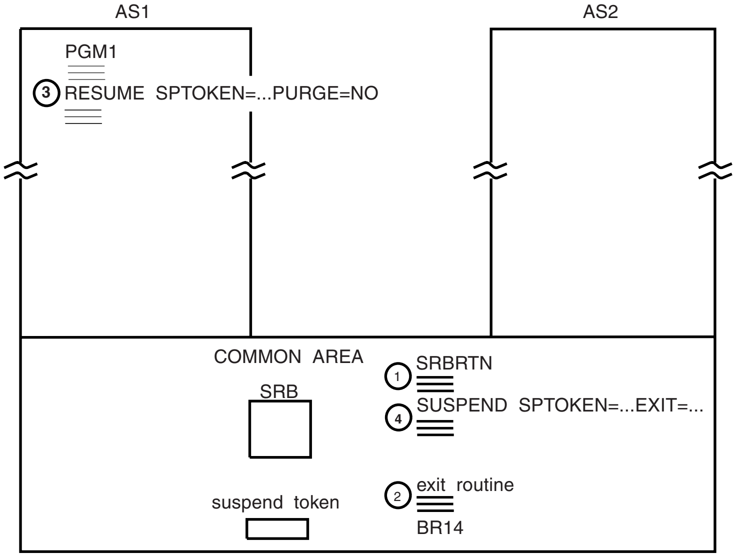


Figure 34. Suspending and Resuming an SRB

When SRBRTN needs the resource, it issues the SUSPEND macro, providing a location where the system can return a suspend token and identifying an exit routine.

The exit routine, which runs under the SRB that represents SRBRTN, receives the suspend token and stores it in the common area. It then queues a request for the resource and returns to the system indicating that the SRB is to remain in a suspended state.

PGM1 receives control when the resource is free. It obtains the suspend token from the common area and issues the RESUME macro, providing the suspend token and specifying PURGE=NO. SRBRTN then resumes control, able to use the resource.

It is important that you understand the order in which the code runs. The circled numbers indicate the order. The SUSPEND macro causes the exit routine to run. The RESUME macro causes the code following the SUSPEND macro to run.

Recovery responsibilities for a suspended SRB

During address space termination, the recovery programs established by suspended SRB routines in that address space do not get control. For this reason, the program that suspends those SRB routines should provide recovery for the SRB routine for the time the SRB routine is suspended. To do this, the resuming or purging program should have used the RESMGR macro to establish a resource manager that gains control should the address space terminate. The resource manager must free any resources owned by the SRB routine and perform recovery, as needed.

When the target of a PURGEDQ macro is a terminating task, the system looks for suspended SRB routines that were scheduled with SRBPTCB and SRBPASID fields that identify the terminating task. The system abnormally ends those SRB routines with an X'47B' system completion code.

When the target of a PURGEDQ macro is the issuer's primary address space, the system looks for preemptable SRB routines that were scheduled with SRBPTCB, SRBPASID, and SRBRMTR fields that match the TCB, ASID, and RMTR parameters on PURGEDQ. If the fields match the parameters, the system abnormally ends the SRB routine, after converting the SRB to be nonpreemptable and nonquiescable, with an X' 47B' system completion code, and waits for termination to complete.

Terminating a preemptable SRB

Preemptable SRBs scheduled via IEAMSCHD with the SRBIDTOKEN parameter can be terminated using CALLRTM TYPE=SRBTERM, SRBIDTOKEN=*token addr*. To temporarily protect an executing preemptable SRB from CALLRTM TYPE=SRBTERM, obtain a lock or set process-must-complete. Note that the system will honor the pending SRBTERM request when your SRB releases its lock or resets process-must-complete.

Calling an SRB to run synchronously

To suspend the caller's thread until the scheduled SRB completes, is purged, or abends, specify the SYNCH=YES parameter on IEAMSCHD. Note that if an SRB scheduled with SYNCH=YES abends, SRB to task percolation does not occur. The calling thread can determine the final status of the scheduled SRB by specifying the SYNCHCOMPADDR= and SYNCHCODEADDR= parameters on IEAMSCHD.

Transferring control for SRB processing (TCTL macro)

The TCTL (transfer control) macro allows an SRB routine to exit from its processing and to pass control to a task with minimal system overhead. When an SRB specifies RESUME RETURN=N, control transfers to the resumed TCB. Control then passes to the top RB on the TCB/RB chain, but only if the system determines that the RB is dispatchable.

Some other considerations for using the TCTL macro are:

- The TCTL macro can be used only by SRB programs, but they may be in any key. If a non-SRB routine issues either the TCTL macro or a RESUME RETURN=N, the routine will abnormally terminate with a X'070' system completion code.
- The TCTL macro causes cleanup of the SRB.
- The TCTL macro requires inclusion of the CVT mapping macro.

The TCTL macro requires that the SRB requesting the TCTL must not hold any locks and must be in primary ASC mode, where the home and primary address space is the same.

Chapter 10. Virtual storage management

The system allocates and releases blocks of virtual storage on request, ensures that real frames exist for SQA, LSQA, and V=R pages, and protects storage with fetch and storage protection keys. It provides this function through the GETMAIN, FREEMAIN, and STORAGE macros. In addition, the system provides the following services through the use of the macros specified:

- VSMREGN macro: List the starting address and the size of the private area regions associated with a given task
- VSMLOC macro: Verify that a given area has been allocated through a GETMAIN or STORAGE macro
- VSMLIST macro: List the ranges of virtual storage allocated in a specified area.

These system services are especially useful when determining available storage, coding recovery procedures, or specifying areas to be included in a dump. VSMREGN enables you to determine the amount of storage that you have for potential use.

z/OS R2 introduces 64-bit addressing and an address space with an almost unlimited amount of virtual storage above the 2-gigabyte address. The size of the 64-bit address space is 16 exabytes, which makes the new address space 8 billion times the size of the former S/390 address space. Programs continue to be loaded and to run below the 2-gigabyte address; these programs can use data that resides above 2-gigabytes. To allocate and release virtual storage above the 2-gigabyte address, a program issues the IARV64 macro. The GETMAIN, FREEMAIN, STORAGE, and CPOOL macros do not allocate storage above the 2-gigabyte address, nor do callable cell pool services. It is expected that most programs will continue to use virtual storage below the 2-gigabyte address. To learn how to obtain and use storage above the 2-gigabyte address, see Using the 64-bit address space in *z/OS MVS Programming: Extended Addressability Guide*.

If you need to check whether a GETMAIN or STORAGE macro was issued to allocate a given block of storage, you can use the VSMLOC macro to perform this check. If the given block is located in private area storage, you can also request the address of the TCB that issued the GETMAIN or STORAGE macro. VSMLOC enables you to verify control blocks or storage locations when coding recovery procedures. You can use VSMLOC to check whether a control block has been allocated and to verify that the control block is located in the correct subpool.

VSMLIST enables you to obtain detailed information about virtual storage that could be useful in determining the areas that you might need in a dump and thereby limit the size of the dump. Limiting the size of a dump is especially critical when executing in 31-bit addressing mode because of the amount of storage involved. The use of VSMLIST is described in “Obtaining information about the allocation of virtual storage (VSMLIST)” on page 232.

You can use the Page Status Interface Routine (IARQD) to determine the location and status of a page in the storage hierarchy. The use of IARQD is described in “Using IARQD — The page status interface routine” on page 240.

Allocating and freeing virtual storage (GETMAIN, FREEMAIN and STORAGE macros)

The GETMAIN macro and the STORAGE macro with the OBTAIN parameter allocate virtual storage. The FREEMAIN macro and the STORAGE macro with the RELEASE parameter free virtual storage.

Most of the functions of GETMAIN, FREEMAIN, and STORAGE are available to all users. You can find information on these functions in *z/OS MVS Programming: Assembler Services Guide*. However, some of the GETMAIN, FREEMAIN, and STORAGE functions are available only to authorized programs. This information provides help in the following areas:

- To help you decide which of the macros to use to obtain and release storage, see "Comparison of GETMAIN/FREEMAIN macros with the STORAGE macro."
- To learn how your program can enter the GETMAIN and FREEMAIN macro service routines through a branch instruction, see "Specifying branch entry to GETMAIN and FREEMAIN services" on page 215.
- To learn how your program can use STORAGE and GETMAIN to obtain storage with a specific storage protection key. See "Selecting the storage key" on page 221.
- To learn how your program can use the STORAGE macro to obtain and release storage in an address space that is not the primary, see "Obtaining storage in another address space" on page 215.
- To learn how to choose the right subpool for your storage request, see "Selecting the right subpool for your virtual storage request" on page 217.
- To learn how your program can associate storage to be obtained with an address space or the system (so the system can track ownership of CSA, ECSA, SQA, and ESQA storage), see "Tracking virtual storage allocation (CPOOL BUILD, GETMAIN, and STORAGE OBTAIN macros)" on page 231.

Comparison of GETMAIN/FREEMAIN macros with the STORAGE macro

The decision about whether to use GETMAIN or STORAGE OBTAIN to obtain virtual storage and FREEMAIN or STORAGE RELEASE to release the storage depends on several conditions:

- **The address space control (ASC) mode of your program**. If it is in AR mode, use the STORAGE macro.
- **The address space that contains the storage your program wants to obtain or release.** If the storage is in an address space other than the primary, use the STORAGE macro.
- **Whether the program requires a branch entry or a stacking PC entry to the macro service**. Using the branch entry on the GETMAIN or FREEMAIN macro is more difficult than using the STORAGE macro. Therefore, you might use STORAGE OBTAIN instead of GETMAIN for ease of coding, for example, when your program:
  - Is in SRB mode
  - Is in cross memory mode
  - Is running with an enabled, unlocked, task mode (EUT) FRR

  The branch entry (BRANCH parameter on GETMAIN or FREEMAIN) requires that your program hold certain locks. STORAGE does not have any locking requirement.

If your program runs in an environment where it can issue the FREEMAIN macro (as specified by the conditions listed above), you can use FREEMAIN to free storage that was originally obtained using STORAGE OBTAIN. You can also use STORAGE RELEASE to release storage that was originally obtained using GETMAIN.

Specifying branch entry to GETMAIN and FREEMAIN services

In addition to SVC entry available to programs in problem state, branch entry is available for authorized programs using the GETMAIN and FREEMAIN macros. Thus, authorized programs that cannot issue SVCs can still use GETMAIN and FREEMAIN.

Two types of branch entry are available for GETMAIN and FREEMAIN: BRANCH=YES and BRANCH=(YES,GLOBAL). BRANCH=YES allows both local (private area) and global (common area) storage to be obtained or released. BRANCH=(YES,GLOBAL) allows only global storage to be obtained or released. BRANCH=(YES,GLOBAL) requests are limited to the following global subpools: The common service area (CSA) subpools 227, 228, 231, and 241, and the system queue area (SQA) subpools 226, 239, 245, 247, and 248.

See the BRANCH parameter descriptions for GETMAIN and FREEMAIN in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for information on how to code the two types of branch entries.

Obtaining storage in another address space

Your program can obtain and release storage in an address space other than the primary address space. Use the ALET parameter on the STORAGE OBTAIN and STORAGE RELEASE macros. To understand access lists and ALETs, and how to get addressability to other address spaces, see *z/OS MVS Programming: Extended Addressability Guide*.

Obtaining and using disabled reference (DREF) storage

A program that runs disabled for I/O or external interrupts cannot reference data-in-virtual storage unless the storage has been page-fixed or is in either a fixed or disabled reference (DREF) subpool.

All storage supported by the system is either fixed, pageable, or disabled reference (DREF). Because the system resolves page faults occurring on DREF storage synchronously, your program can reference DREF storage while it runs disabled for I/O and external interrupts. Your program, therefore, can use DREF storage in place of fixed storage if it needs to reference storage while it is disabled. An advantage of DREF storage is that it need not be backed by central storage frames until it is referenced. However, if the system cannot obtain a frame of central storage to back the DREF storage when it is referenced, the program referencing the storage is ended abnormally.

DREF storage is similar to pageable storage in the following ways:

- It cannot be referenced with DAT off
- It cannot be used for I/O
- Its virtual addresses do not map to constant real addresses.

IBM recommends that you use pageable storage instead of DREF storage if possible. You should also consider using the PGSER macro to page-fix pageable storage as needed.

To obtain storage from a DREF subpool, use the SP parameter on either GETMAIN or STORAGE and specify subpool 203, 204, 205, 213, 214, 215, 247, or 248. These subpools are the DREF subpools.

Avoid using the load real address (LRA) instruction to get the real address of DREF storage. Do not reference DREF storage with an I/O operation or with dynamic address translation (DAT) turned off. Also, because the DREF subpools are supported only above 16 megabytes, specify LOC=31 on the GETMAIN macro or STORAGE OBTAIN macro.

You can also use the DSPSERV macro to obtain a data space of DREF storage. The *z/OS MVS Programming: Extended Addressability Guide* contains information on how to create a data space with DREF storage.

Abends while referencing DREF storage: If your program attempts to reference DREF storage and the system cannot obtain central storage to back it, the system will abnormally terminate your program with an abend code of X'03C'. If you plan to use DREF storage, your program should be prepared to handle such an abend. Furthermore, your program's recovery routine should refrain from referencing any DREF storage if that is what caused the abend.

Using cell pool services (CPOOL macro)

The cell pool macro, CPOOL, provides users with another way of obtaining virtual storage. This macro provides centralized, high performance cell management services.

Cell pool services obtain a block of virtual storage (called a cell pool) from a specific subpool at the user's request. The user can then request smaller blocks of storage (called cells) from this cell pool as needed. If the storage for the requested cells exceeds the storage available in the cell pool, the user can also request that the cell pool be increased in size (extended) to fill all requests.

The CPOOL macro makes the following cell pool services available:

- Create a cell pool (BUILD)
- Obtain a cell from a cell pool if storage is available (GET,COND)
- Obtain a cell from a cell pool and extend the cell pool if storage is not available (GET,UNCOND)
- Return a cell to the cell pool (FREE)
- Free all storage for a cell pool (DELETE)
- List the beginning address and ending address of every extent in a cell pool (LIST)

The CPOOL macro, with the exception of the TCB, VERIFY, and LINKAGE=BRANCH parameters, is available to unauthorized users. If unauthorized programs specify the OWNER parameter, the system ignores the parameter (because unauthorized programs cannot obtain common storage). Note, however, that in order to provide high performance, cell pool services do not attempt to detect most user errors. For example, the following user errors are not detected by cell pool services, and may produce unpredictable results:

- The user is executing in a non-zero key that does not match the key of the pool being manipulated.
- The user attempts to free a cell from a pool that has already been deleted.

- When trying to free a cell, the user passes cell pool services a bad cell address. (This might damage the cell pool, preventing subsequent requests from being properly handled.)
- A disabled user requests that a cell pool be built in a pageable subpool.

Your program can use the LIST service when it wants to know the starting and ending addresses of cell pool extents. It can use this information when it requests SVC dumps of cell pool storage. Because of the limit of the input buffer, the system might not be able to complete the request for the listing in one issuance of CPOOL LIST. It can reissue the macro. In the first bit of the work area it supplies on the WORKAREA parameter, the program must tell the system whether this is the first issuance of the macro. A “1” in the first bit indicates the first time the macro is issued and a “0” indicates subsequent issuances. The system returns a code to let your program know when it has more information to give and when it has completed the request. Return codes are listed in *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*. Do not issue any other CPOOL requests between the issuances of CPOOL LIST.

Selecting the right subpool for your virtual storage request

Conceptually, a subpool is an area of virtual storage with a certain set of attributes, created by MVS to satisfy a request for virtual storage. A two-gigabyte virtual storage address space is divided into multiple areas, each intended to support specific system and user needs. MVS provides subpools within the following areas of the address space:

- Low private
- High private
- Private local system queue area (LSQA) and extended LSQA
- Common system queue area (SQA) and extended SQA
- Common service area (CSA) and extended CSA.

For more information about these virtual storage areas as well as information about virtual storage in general, see the virtual storage overview in *z/OS MVS Initialization and Tuning Guide*.

Besides being in a particular area in an address space, subpools are distinguished by other attributes, such as which programs can access them, whether the storage can be paged out, and how long the storage persists. To select the right subpool, you need to examine these attributes and determine which ones are best suited for your needs. Your choice of subpool depends on several factors:

- Whether your program is authorized
- How you will use the storage
- What environment your program runs in.

This information discusses the considerations involved in determining the appropriate subpool for your storage request, such as:

- What is your program's authorization level? (See “Program authorization” on page 218.)
- Do other address spaces need access to the storage or will it be used only by tasks and subtasks in the same address space? (See “Selecting private or common storage” on page 219.)
- Do you want to protect the storage area from unwanted access? (See “Selecting fetch protected or non-fetch protected storage” on page 220.)

- Does your storage area require a particular storage key? (See “Selecting the storage key” on page 221.)
- Is the program accessing the storage running enabled, locked, or disabled? (See “Selecting pageable, DREF, or fixed storage” on page 222.)
- How long will the storage be required? (See “Selecting storage persistence” on page 223.)
- Must the storage area be backed below 16 megabytes or can it be backed anywhere? (See “Selecting central storage backing” on page 226.)

Be sure also to read the discussion of subpools in *z/OS MVS Programming: Assembler Services Guide*. The information in *z/OS MVS Programming: Assembler Services Guide* is not repeated in this discussion.

Alternatives to Using Subpool Storage

Data spaces and hiperspaces provide an alternative means of obtaining and sharing storage. For instance, you can use a SCOPE=ALL data space to hold data to be accessed by all programs running in your address space. You can also use a SCOPE=COMMON data space to share data among programs in different address spaces instead of using common storage for that purpose. For more information about creating and using data spaces and hiperspaces, see *z/OS MVS Programming: Extended Addressability Guide*.

Program authorization

Your program's authorization determines what subpools you can use. For the GETMAIN, FREEMAIN, STORAGE, and CPOOL macros, your program is authorized if it has at least one of the following types of authorization:

- Runs in supervisor state
- Has PSW key 0-7
- Is APF-authorized

Otherwise, the program is unauthorized. Unauthorized programs can request storage only from private storage subpools 0-127, 131, and 132. Authorized programs can obtain storage from additional private storage subpools as well as from common storage subpools.

Obtaining and releasing storage from subpool 0 yields different results depending on whether a program is running in supervisor state and PSW key 0. If a program is running in supervisor state and PSW key 0, the system translates the subpool 0 storage request to a subpool 252 storage request. In all other cases, the system handles subpool 0 storage requests as such. To obtain or free subpool 0 storage while running in supervisor state and PSW key 0, a program must specify subpool 240 or 250 on the storage request. Subpool 240 and 250 requests, which are limited to authorized callers, are translated to subpool 0 requests. Table 26 summarizes these special cases.

Table 26. Supervisor State and PSW Key 0 Callers and Subpool 0

Subpool Specified	Subpool Obtained or Freed
240, 250	0
0	252

Because of the way the system translates requests from supervisor state and PSW key 0 callers involving certain subpools, a problem such as the following could occur. A program running in supervisor state and PSW key 0 switches into another PSW key and obtains storage from subpool 0. The program then switches back to

PSW key 0 and tries to free the storage. An error results because the system translates the request to free storage from subpool 0 to a request to free storage from subpool 252. To prevent this error, the program must specify subpool 240 or 250 when releasing the subpool 0 storage.

Selecting private or common storage

Your choice of private or common storage depends on whether programs in other address spaces need access to your storage.

Private storage is divided into **low private** and **high private** storage. The system satisfies low private storage requests by allocating storage at the lowest available address. The system satisfies high private storage requests by allocating storage at the highest available address. This method of allocating low and high private storage allows the amount of low and high private to adjust to the needs of the address space. Certain considerations apply to allocating storage within the private area. See “Managing private storage allocation” for further information.

Low private storage can be obtained and released by all programs; in contrast, high private storage can be obtained and released only by authorized programs. Common storage can be read by both authorized and unauthorized programs, but can be obtained and released only by authorized programs. Each address space in the system has its own private storage but shares common storage with the other address spaces. Although common storage is considered part of an address space, in many ways it behaves as if it were not. For instance, if an address space is swapped out, only the private storage is swapped out. Private storage can be accessed from another address space only by using access registers or cross-memory services; whereas common storage can be accessed by tasks and SRBs in other address spaces.

Note: Private and **common** storage are frequently referred to as **local** and **global** storage, respectively.

Table 27 summarizes the differences between private and common storage:

Table 27. A Comparison of Private and Common Storage

Private Storage	Common Storage
Each address space has its own	All address spaces share
Accessible only using the dynamic address translation (DAT) tables for that address space	Accessible using the DAT tables for any address space
Swappable if address space is swappable	Non-swappable
Amount of low private storage limited by the user region size, an installation-defined parameter. For more information about the user region, see *z/OS MVS Initialization and Tuning Guide*.	Amount limited by sizes of SQA and CSA as defined in the IEASYSxx member of SYS1.PARMLIB. For more information about the IEASYSxx member, see *z/OS MVS Initialization and Tuning Reference*.

Managing private storage allocation

When the system creates an address space, it assigns a certain area of the address space for use as private storage. High private storage is allocated from the top of the private area down, and low private storage is allocated from the bottom of the private area up. Private storage allocation is governed by two rules:

1. Low private storage cannot exceed the boundary set by the installation-determined user region size limit.

2. High private can "grow" down as long as it does not go below the low storage boundary, determined by the allocated low private storage area at the highest address. That is, high private storage must be allocated above all allocated low private storage, and low private storage must be allocated below all allocated high private storage.

High private storage can be allocated below the user region boundary as long as the high private storage resides above all allocated low private storage.

As a result of rule 2, several problems could occur. First, a caller might be unable to obtain low private storage even though low private storage is available because an area of high private storage is in the way and the available storage is above it. Figure 35 illustrates the problem.

To reduce the possibility of being unable to obtain low private storage, try to obtain storage areas for long-term use first so that they will be allocated at the top (high private storage) or bottom (low private storage) of the private area, preventing long-term allocations from fragmenting private storage. You should also release any storage when it is no longer needed.

Another problem could occur due to the allocation of large amounts of high private storage. The amount of low private storage could be reduced to the point that the system cannot provide a user region of the specified size when it attempts to a start a new step. If this situation occurs, the step is ended abnormally.

To minimize this problem, try to avoid fragmenting high private storage because this causes successive allocations to reach farther and farther down into the low private area. You should also release any storage when it is no longer needed.

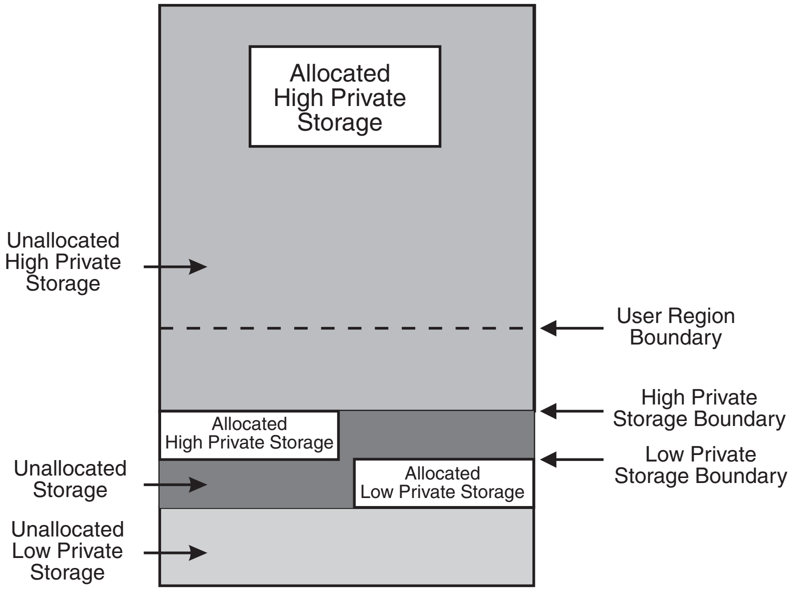

Allocated High Private Storage
Unallocated High Private Storage
User Region Boundary
High Private Storage Boundary
Allocated High Private Storage
Low Private Storage Boundary
Unallocated Storage
Allocated Low Private Storage
Unallocated Low Private Storage

Figure 35. Low and High Private Storage Allocation

Selecting fetch protected or non-fetch protected storage

Fetch protection helps prevent unwanted read access to your storage. A program can read from fetch protected storage only if its PSW key matches the storage key of the storage area or if its PSW key is 0. Non-fetch protected storage can be read by any program.

Note: Although you can obtain fetch-protected storage in key 8, the storage is only protected from accidental access because any program can switch into PSW key 8 and thereby obtain access to key-8 storage.

Selecting the storage key

The storage key helps control read and write access to the virtual storage area. You need to choose a storage key that will allow access by those programs that require it. To limit unwanted access by other programs, try to select a storage key different from the PSW key that other programs are likely to use.

The storage key controls user access as follows:

- Programs running in PSW key 0 have read and write access to storage in every storage key.
- Programs running in PSW keys 1 through 15 have read access to:
  - Storage whose key matches their PSW key
  - Storage (in any key) that's not fetch protected
  - Storage in key 9 if the hardware feature, subsystem storage protection override, is installed. The CVTOVER bit in the CVT is set to 1 if the feature is installed. For more information about the CVTOVER bit in the CVT, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).
- Programs running in PSW keys 1 through 15 have write access to:
  - Storage whose key matches their PSW key
  - Storage in key 9, if subsystem storage protection override is installed.

The amount of control you have over the storage key assigned to your virtual storage request varies depending on the subpool from which you request storage. For the purpose of storage key assignment, there are three types of subpools. The first type of subpool is always assigned a particular storage key, such as 0. The second type of subpool is assigned its storage key based on the key in the requesting task's TCB at the time of that task's first storage request. The third type of subpool allows you to select the storage key by:

- Specifying the KEY parameter on the STORAGE, GETMAIN, or FREEMAIN macros, or the CPOOL macro with the BUILD parameter.
- Choosing a particular macro and set of parameters for the storage request. (The combination of macro and parameters determines the key assigned to the storage request for these subpools.)
- Specifying CALLRKY=YES on the STORAGE macro, which causes the storage key to match the calling program's current PSW key.

These subpools are shown in the subpool table (Table 32 on page 228) as having a selectable storage key. Detailed information about subpools with selectable storage keys is provided in Table 28.

Table 28. Storage Keys for Selectable Key Subpools

Subpool	Macros and Parameters	Storage Key
129-132	• GETMAIN with LC, LU, VC, VU, EC, EU, or R; BRANCH not specified • FREEMAIN with LC, LU, L, VC, VU, V, EC, EU, E, or R; BRANCH not specified • STORAGE with OBTAIN or RELEASE; CALLRKY=YES is specified	The storage key equals the caller's PSW key. (The KEY parameter is not allowed.)

Table 28. Storage Keys for Selectable Key Subpools (continued)

Subpool	Macros and Parameters	Storage Key
129-132	• GETMAIN with LC, LU, VC, VU, EC, EU, or R; BRANCH=YES specified • FREEMAIN with LC, LU, L, VC, VU, V, EC, EU, E, or R; BRANCH=YES specified	The storage key is 0. (The KEY parameter is not allowed.)
129-132	• GETMAIN with RC, RU, VRC, VRU; BRANCH not specified • FREEMAIN with RC, RU; BRANCH not specified • CPOOL with BUILD	The storage key is the key the caller specifies on the KEY parameter. If KEY is not specified, the default equals the caller's PSW key.
129-132	• GETMAIN with RC, RU, VRC, VRU; BRANCH=YES specified • FREEMAIN with RC, RU; BRANCH=YES specified • STORAGE with OBTAIN or RELEASE; CALLRKY=YES is omitted, or CALLRKY=NO is specified	The storage key is the key the caller specifies on the KEY parameter. If KEY is not specified, the default is 0.
227-231, 241, 249	• All GETMAIN requests with BRANCH not specified • All FREEMAIN requests with BRANCH not specified • STORAGE with OBTAIN or RELEASE; CALLRKY=YES specified	The storage key equals the caller's PSW key. (For RC, RU, VRC, and VRU, the KEY parameter is ignored. For other GETMAIN and FREEMAIN requests, the KEY parameter is not allowed.)
227-231, 241, 249	• GETMAIN with LC, LU, VC, VU, EC, EU, or R; BRANCH=YES specified • FREEMAIN with LC, LU, L, VC, VU, V, EC, EU, E, or R; BRANCH=YES specified	The storage key is 0 (The KEY parameter is not allowed.)
227-231, 241, 249	• GETMAIN with RC, RU, VRC, VRU; BRANCH specified **Note:** BRANCH=(YES,GLOBAL) is not valid for subpools 229, 230, and 249. • FREEMAIN with RC, RU; BRANCH specified **Note:** BRANCH=(YES,GLOBAL) is not valid for subpools 229, 230, and 249. • STORAGE with OBTAIN or RELEASE; CALLRKY=YES omitted, or CALLRKY=NO specified	The storage key is the key the caller specifies on the KEY parameter. If KEY is not specified, the default is 0.
227-231, 241, 249	• CPOOL with BUILD	The storage key is the key the caller specifies on the KEY parameter. If KEY is not specified, the default equals the caller's PSW key.

Selecting pageable, DREF, or fixed storage

Whether you choose pageable, DREF, or fixed storage depends on the extent to which your program is running disabled for external and I/O interrupts. To determine the type of storage you require, use the following guidelines:

- If your program runs enabled for external and I/O interrupts, use pageable storage. If your program requires fixed storage for only a short interval, use the PGSER macro to page-fix the storage area as needed.
- If your program meets the following criteria, consider using DREF storage:
  - Runs disabled for external and I/O interrupts
  - Has no requirement for constant real addresses backing virtual addresses
  - Is authorized to obtain storage in storage key 0.

  Attention: Before choosing DREF storage, read “Obtaining and using disabled reference (DREF) storage” on page 215, for information about special recovery requirements for its use.
- If your program runs disabled for external and I/O interrupts for the duration of its processing and requires constant real addresses backing virtual addresses,

use fixed storage. Note that the storage area must be non-swappable (either located in common storage or in a non-swappable address space) to have fixed real addresses.

You should select the type of storage that meets your needs while minimizing your use of central storage. Central storage is the most limited resource, followed by expanded storage, followed by auxiliary storage.

Pageable and DREF storage are backed by central storage only when the virtual storage is referenced. Fixed storage is always backed by central storage.

Selecting storage persistence

To select the storage persistence, you must decide how long you require the storage. Before making these decisions, you need to understand what determines the persistence of a virtual storage area.

The level of persistence is determined by whether the storage is owned by a task, a job step task, an address space, or the system. **When the owner terminates, the system releases the storage**. In the subpool table in Table 32 on page 228, the "owner" heading shows the owner of each subpool. The four categories are:

Task The storage is owned by a task's TCB.

Job step
The storage is owned by a job step task's TCB.

Address space
The storage is owned by the address space in which the storage resides.

System
The storage is owned by the system; it is not associated with a particular task or address space.

Note: Storage ownership as it is discussed here is unrelated to the OWNER keyword on the STORAGE, GETMAIN, and CPOOL macros.

To understand how the system determines the ownership of task and job step owned storage, you need to understand the concept of the input TCB. The **input TCB** is the TCB whose address is specified either explicitly or by default on a request to obtain or free private storage. The system uses the input TCB to assign ownership of the storage being obtained or to determine the ownership of the storage being freed. Note that the system does not necessarily assign storage ownership to the input TCB:

- **For task owned storage: the owning TCB is the input TCB**.
- **For job step owned storage: the owning TCB is the job step task TCB of the input TCB**.

Note: The rules for task and job step owned storage ownership are different for shared subpools. For information about how the system assigns ownership for storage obtained from shared subpools, see “Additional considerations” on page 227.

Task owned storage: For task owned storage, the input TCB is determined by the macro used to obtain or free the storage, the parameters specified, the dispatchable unit mode of the caller, and the address space in which the storage is obtained (the target address space). If the calling program is running in SRB mode or obtaining storage in an address space other than the home address space, the system assigns the input TCB to be the TCB that owns the cross memory resources in the address space in which the storage is being obtained. The address of this TCB is stored in the ASCBXTCB field of the ASCB of that address space. Table 29 shows how the system determines the input TCB for task-owned private storage.

Table 29. How the System Determines the Input TCB for Task Owned Storage

Macro	Conditions	Input TCB	Event Causing Storage to be Freed
GETMAIN and FREEMAIN macros	If the caller specifies SVC entry	TCB of the currently active task, whose address is in PSATOLD	Currently active task terminates
GETMAIN and FREEMAIN macros	If the caller specifies local branch entry and specifies GPR 4 with a value of zero	TCB owning the cross-memory resources in the target address space, whose address is in ASCBXTCB	Task, whose TCB address is in ASCBXTCB, terminates
GETMAIN and FREEMAIN macros	If the caller specifies local branch entry and specifies GPR 4 with a non-zero value	TCB address specified by the caller in GPR 4	Task, whose TCB address is passed in GPR 4, terminates
STORAGE macro	If the caller is in task mode and the target address space is the home address space	TCB of the currently active task, whose address is in PSATOLD	Currently active task terminates
STORAGE macro	If the caller is in SRB mode or the target address space is not the home address space	TCB owning the cross-memory resources in the target address space, whose address is in ASCBXTCB	Task, whose TCB address is in ASCBXTCB, terminates
STORAGE macro	If the caller specifies the TCBADDR parameter	TCB specified by the caller with the TCBADDR parameter	Task whose TCB address is specified with the TCBADDR parameter terminates
CPOOL macro	If the caller omits the TCB parameter	TCB of the currently active task, whose address is in PSATOLD	Currently active task terminates
CPOOL macro	If the caller specifies the TCB parameter with a value of 0	TCB owning the cross-memory resources in the target address space, whose address is in ASCBXTCB	Task, whose TCB address is in ASCBXTCB, terminates
CPOOL macro	If the caller specifies the TCB parameter with a non-zero value	TCB specified by the caller with the TCB parameter	Task, whose TCB address is specified with the TCB parameter, terminates

Job step owned storage: For job step owned storage, the owning TCB is the job step task TCB of the input TCB. That is, the owning TCB is the job step task TCB of the TCB that would own the storage according to Table 29.

Figure 36 on page 225 shows an example of a task structure within an address space, including the locations of the ASCBXTCB and TCBJSTCB fields. Note that the task structure within an address space can vary and this example only shows one of many different possibilities.

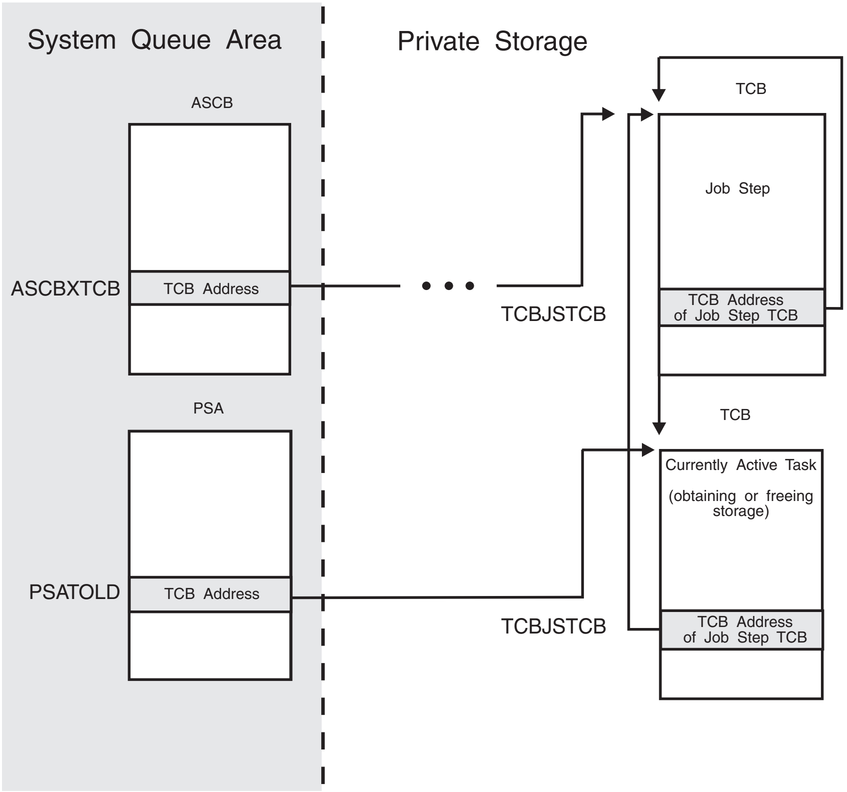


Figure 36. Task Structure Within an Address Space

Address space owned storage: Address space owned storage is private storage that is owned by the address space in which it resides. It is not associated with a task.

System owned storage: System owned storage persists until explicitly freed using the FREEMAIN macro, the STORAGE macro with the RELEASE parameter, or the CPOOL macro with the DELETE parameter. If an area of system owned storage is not freed, it remains allocated for the life of the IPL. System owned storage is not associated with a task or an address space. Common storage subpools are system owned. Private storage subpools are all task, job step task, or address space owned.

The important consideration in selecting the type of storage ownership is to ensure that the storage you obtain remains in existence for the required duration. Because the system automatically frees all non-system-owned storage when the owner terminates, you must ensure that storage ownership is assigned to an entity that will not terminate while the obtained storage area is still needed.

You should always ensure that storage explicitly obtained is explicitly freed. It is not a good programming practice to rely on the system to free storage for you, except in cases where there is no alternative.

Now that you understand the types of storage ownership and how storage ownership is assigned, use the following guidelines to help you determine the type of storage ownership you need:

- If you require the storage only during your program's processing, use task owned storage.
- If you require the storage to persist after your program has terminated but not after the owning job step task has terminated, use job step owned storage. For

instance, it is necessary to have storage persist after your program has terminated if your program obtains storage that must be accessed by another task that will be running after your program terminates.

- If you require the storage to persist for the life of the address space, use address space owned storage. This is necessary, for instance, if the storage must be available to job step tasks running later in the same address space.
- If you require the storage to persist after the address space has terminated, use system owned storage (which is exclusively common storage). This is necessary, for instance, if the storage must be available permanently to all address spaces.

Table 30 lists the types of storage ownership and the circumstances under which each is appropriate.

Table 30. Choosing Storage Ownership

If you need the storage:	**Storage should be owned by:**
Only during your program's processing	Task
After your program terminates but not after the job step task has terminated	Job step
For the life of the address space but not after the address space has terminated	Address space
After the address space has terminated	System

Selecting central storage backing: Virtual storage is first backed by central storage when it is:

- Referenced
- Obtained from a fixed storage subpool
- Page-fixed by a program using the PGSER macro
- Obtained from a pageable storage subpool (only in some cases).

When the storage is fixed, either implicitly through the subpool attribute, or explicitly through PGSER FIX, the LOC parameter, specified on the GETMAIN STORAGE, or CPOOL macro invocation used to obtain the storage, determines where the storage will be backed in central storage. Storage which is not fixed can be backed anywhere in central storage.

The STORAGE OBTAIN PAGEFRAMESIZE1MB option specifies to back storage by 1MB large page frames, if available.

The system will back the storage below 16 megabytes only if one of the following is true:

- You specify LOC=24 when obtaining storage and the subpool is supported below 16 megabytes, and the subpool is not an SQA or LSQA subpool.
- You specify LOC=RES either explicitly or by default, your program resides below 16 megabytes, and the subpool is supported below 16 megabytes, and the subpool is not an SQA or LSQA subpool.
- You specify subpool 226, which is always backed below 16 megabytes.

Otherwise, the system might back the virtual storage request with central storage above 16 megabytes.

You should not need central storage backing below 16 megabytes unless your program requires 24-bit real addresses.

The system may back fixed virtual storage above 2 gigabytes if the subpool is not 226, and you specify one of the following:

- LOC=(24,64)
- LOC=(RES,64)
- LOC=(31,64)
- LOC=(31,PAGEFRAMESIZE1MB)
- LOC=(EXPLICIT,64)
- LOC=(EXPLICIT,PAGEFRAMESIZE1MB)

Additional considerations: In addition to the previously described considerations for selecting a virtual storage subpool, you should keep in mind the following:

- If your program obtains storage from a subpool shared between a task and its attached subtasks, private storage ownership is assigned as follows:
  - For task owned storage: the owning task is the task that attached the subtasks.
  - For job step task owned storage: the owning task is the job step task of the task that attached the subtasks.

  The storage key associated with the shared subpool is determined at the time storage is first obtained from that subpool. All subsequent storage requests filled from that subpool have that storage key. For more information about shared subpools, see "Virtual Storage Management" in *z/OS MVS Programming: Assembler Services Guide*.
- If you are allocating a very large storage area, you might want to read *z/OS MVS Diagnosis: Tools and Service Aids* to find out what subpools are dumped by default during a stand-alone dump. If the data to be stored in the storage area would not be of interest in a stand-alone dump and the storage area is large (requires a long time to dump), you should obtain storage from a subpool that is not dumped by default.

Making the final selection: After determining the storage attributes necessary for your storage request, use Table 31, which shows subpools organized by attribute and owner, to obtain candidate subpool numbers. Then use the subpool table, Table 32 on page 228, to obtain additional information about each candidate subpool before making your final selection.

Table 31. Subpools Grouped by Attributes

		Task Owned	Job Step Owned	Address Space Owned	System Owned
Private, Fetch Protected	Pageable	0-127, 229, 240, 250	129, 131, 251	None	None
Private, Fetch Protected	DREF	213	214	215	None
Private, Fetch Protected	Fixed	223	224	225	None
Private, Not Fetch Protected	Pageable	230, 236, 237	130, 132, 249, 244, 252	None	None
Private, Not Fetch Protected	DREF	203	204	205	None
Private, Not Fetch Protected	Fixed	233, 253	234, 254	235, 255	None
Common, Fetch Protected	Pageable	None	None	None	231
Common, Fetch Protected	DREF	None	None	None	247
Common, Fetch Protected	Fixed	None	None	None	227, 239

Table 31. Subpools Grouped by Attributes (continued)

		Task Owned	Job Step Owned	Address Space Owned	System Owned
Common, Not Fetch Protected	Pageable	None	None	None	241
Common, Not Fetch Protected	DREF	None	None	None	248
Common, Not Fetch Protected	Fixed	None	None	None	226, 228, 245

The subpool table: The following table lists the subpools and their attributes. Be sure to read the notes at the end of the table where applicable.

Table 32. Storage Subpools and Their Attributes

Subpool	Location	Fetch Protection	Type	Owner	Storage Key	See Notes at End of Table
0-127	Private low	Yes	Pageable	Task. TCB identified in note 10.	Same as TCB key at time of first storage request.	1,5,7,9,12
129	Private low	Yes	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB identified in note 10.	Selectable. See Table 28 on page 221.	1,12
130	Private low	No	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB identified in note 10.	Selectable. See Table 28 on page 221.	1,12
131	Private low	Yes	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB identified in note 10.	Selectable. See Table 28 on page 221.	1,5,6,12
132	Private low	No	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB identified in note 10.	Selectable. See Table 28 on page 221.	1,5,6,12
203	Private ELSQA	No	DREF	Task. TCB shown in Table 29 on page 224.	0	2,4
204	Private ELSQA	No	DREF	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	0	2,4
205	Private ELSQA	No	DREF	Address space	0	2,4
213	Private ELSQA	Yes	DREF	Task. TCB shown in Table 29 on page 224.	0	2,4
214	Private ELSQA	Yes	DREF	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	0	2,4
215	Private ELSQA	Yes	DREF	Address space	0	2,4
223	Private ELSQA	Yes	Fixed	Task. TCB shown in Table 29 on page 224.	0	2,4
224	Private ELSQA	Yes	Fixed	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	0	2,4
225	Private ELSQA	Yes	Fixed	Address space	0	2,4
226	Common SQA/ESQA	No	Fixed	System	0	3
227	Common CSA/ECSA	Yes	Fixed	System	Selectable. See Table 28 on page 221.	1
228	Common CSA/ECSA	No	Fixed	System	Selectable. See Table 28 on page 221.	1
229	Private high	Yes	Pageable	Task. TCB shown in Table 29 on page 224.	Selectable. See Table 28 on page 221.	1

Table 32. Storage Subpools and Their Attributes (continued)

Subpool	Location	Fetch Protection	Type	Owner	Storage Key	See Notes at End of Table
230	Private high	No	Pageable	Task. TCB shown in Table 29 on page 224.	Selectable. See Table 28 on page 221.	1
231	Common CSA/ECSA	Yes	Pageable	System	Selectable. See Table 28 on page 221.	1
233	Private LSQA/ELSQA	No	Fixed	Task. TCB shown in Table 29 on page 224.	0	2
234	Private LSQA/ELSQA	No	Fixed	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	0	2
235	Private LSQA/ELSQA	No	Fixed	Address space	0	2
236	Private high	No	Pageable	Task. TCB identified in note 11.	1	2
237	Private high	No	Pageable	Task. TCB identified in note 11.	1	2
239	Common SQA/ESQA	Yes	Fixed	System	0	2
240	Private low	Yes	Pageable	Task. TCB identified in note 10.	Same as TCB key at time of first storage request.	1,8,9,12
241	Common CSA/ECSA	No	Pageable	System	Selectable. See Table 28 on page 221.	1
244	Private low	No	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB identified in note 10.	Selectable. See Table 28 on page 221.	1,12
245	Common SQA/ESQA	No	Fixed	System	0	2
247	Common ESQA	Yes	DREF	System	0	2,4
248	Common ESQA	No	DREF	System	0	2,4
249	Private high	No	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	Selectable. See Table 28 on page 221.	1
250	Private low	Yes	Pageable	Task. TCB identified in note 10.	Same as TCB key at time of first storage request.	1,8,9,12
251	Private low	Yes	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	Same as TCB key at time of first storage request.	1,9,12
252	Private low	No	Pageable	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	0	1,12
253	Private LSQA/ELSQA	No	Fixed	Task. TCB shown in Table 29 on page 224.	0	2
254	Private LSQA/ELSQA	No	Fixed	Job step. TCB whose address is in TCBJSTCB of TCB shown in Table 29 on page 224.	0	2
255	Private LSQA/ELSQA	No	Fixed	Address space	0	2

Notes:

1. Virtual storage is first backed by central storage when it is referenced or when it is page-fixed by a program using the PGSER macro. The location of the central storage backing this subpool depends on the value of the LOC parameter on the GETMAIN, STORAGE, or CPOOL macro invocation used to obtain the storage. Central storage is assigned below 16 megabytes only if one of the following is true:

- The program obtaining the storage specified LOC=24 when obtaining the storage.
- The program obtaining the storage resides below 16 megabytes, specified LOC=RES either explicitly or by default, and specified a subpool supported below 16 megabytes.

2. Central storage backing this subpool can be above 2 gigabytes when LOC=(xx,64) is specified.
3. Central storage backing this subpool resides below 16 megabytes.
4. This subpool is valid only when allocating virtual storage above 16 megabytes.
5. Subpools 0-127, 131, and 132 are the only valid subpools for unauthorized programs. A request by an unauthorized program for a subpool other than 0-127, 131, or 132 causes abnormal termination of the program.
6. A program can issue a request to obtain or release storage from subpool 131 or 132 in a storage key that does not match the PSW key under which the program is running. However, the system will accept the storage request only if the requesting program is authorized in one of the following ways:
   - Running in supervisor state
   - Running under PSW key 0-7
   - APF-authorized (valid for GETMAIN, FREEMAIN, CPOOL, and STORAGE macros)
   - Having a PSW-key mask (PKM) that allows it to switch its PSW key to match the storage key of the storage specified. On a request to release all the storage in the subpool, the program must be able to switch its PSW key to match all the storage keys in the subpool.

   For information about the function and structure of the PKM, and information about switching the PSW key, see *Principles of Operation*.
7. Subpool 0 requests by programs in supervisor state and PSW key 0 are translated to subpool 252 requests and assigned a storage key of 0.
8. Subpool 240 and 250 requests are translated to subpool 0 requests. This permits programs running in supervisor state and PSW key 0 to acquire or free subpool 0 storage. If a program is running in supervisor state and key 0, the system translates subpool 0 storage requests to subpool 252 storage requests. For more information, see Table 26 on page 218.
9. The system assigns the storage key based on the key in the requesting task's TCB at the time of the task's first storage request, not the current key in the TCB (unless this is the task's first storage request).
10. The GSPV, SHSPV, and SZERO parameters on the ATTACH or ATTACHX macro invocation used to create the currently active task determine which TCB owns the storage in this subpool. These parameters specify the subpools to be owned by the subtask being attached (GSPV) and the subpools to be shared by the attaching task and the subtask being attached (SHSPV, SZERO). If the currently active task was given ownership of the subpool, then the TCB of the currently active task owns the storage in this subpool. If the currently active task is sharing this subpool with the task that created it, then the TCB of the attaching task owns the storage in this subpool. For more information, see the descriptions of the ATTACH and ATTACHX macros in *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* and the virtual storage management topic in *z/OS MVS Programming: Assembler Services Guide*.
11. Virtual storage is located within the scheduler work area (SWA). The storage is freed at the end of the started task or at initiator termination for subpool 236 and at the end of the job for subpool 237. The NSHSPL and NSHSPV

parameters on the ATTACH or ATTACHX macro invocation used to create the currently active task determine ownership of the subpool. If the currently active task was given ownership of the subpool, then the TCB of the currently active task owns the storage in this subpool. If the currently active task is sharing this subpool with the attaching task, then the TCB of the attaching task owns the storage in this subpool. For more information, see the description of the ATTACH and ATTACHX macros in *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* and the virtual storage management topic in *z/OS MVS Programming: Assembler Services Guide*. For additional information about the SWA, see *z/OS MVS Initialization and Tuning Guide*.

12. This subpool is backed by a 1 megabyte page frame, if available, when LOC=(31,PAGEFRAMESIZE1MB) or LOC=(EXPLICIT,PAGEGRAMESIZE1MB) is specified on the STORAGE macro.

Tracking virtual storage allocation (CPOOL BUILD, GETMAIN, and STORAGE OBTAIN macros)

Common storage tracking is an optional function that collects information about requests to obtain and free storage in CSA, ECSA, SQA, and ESQA. You can use Resource Measurement Facility (RMF) as of release 4.3 or any compatible monitor program to display the information that the storage tracking function collects. You can also use the interactive problem control system (IPCS) to format the information from a dump.

You can use the RMF or IPCS reports to identify jobs or address spaces that use an excessive amount of storage or have ended without freeing storage. If those jobs or address spaces have code to free that storage when they are canceled, you might relieve the shortage and avoid an IPL if you cancel the jobs or address spaces using an operator command.

The OWNER parameter on the CPOOL BUILD, GETMAIN, and STORAGE OBTAIN macros specifies the entity to which the system will assign ownership of requested CSA, ECSA, SQA, and ESQA storage, which is one of the following:

- An address space that, upon termination, is responsible for freeing the storage.
- The system (the storage is not associated with an address space); specify this value if you expect the requested storage to remain allocated after termination of the job that obtained the storage.

For the storage tracking function to produce accurate results, authorized programs must include the OWNER parameter when issuing CPOOL BUILD, GETMAIN, and STORAGE OBTAIN macros that:

- Request storage in CSA, ECSA, SQA, or ECSA, *and*
- Have an owning address space that is *not* the home address space.

You do not have to code the OWNER parameter when the home address space is to own the requested storage (the default value is HOME).

For information about how to use IPCS to format data collected by the storage tracking function from a dump, see the description of the VERBEXIT VSMDATA subcommand in *z/OS MVS IPCS Commands*.

Obtaining information about the allocation of virtual storage (VSMLIST)

The VSMLIST macro provides information about the allocation of virtual storage. The VSMLIST service routine returns the information in a user-supplied work area specified as a parameter of the VSMLIST macro. The length of the work area varies but it must be a minimum length of 4K bytes.

Using the VSMLIST work area

Prior to the first invocation of the VSMLIST macro for a single request, you must set the first four bytes of the work area to zero. This field will contain the return code of the VSMLIST macro after control returns to the issuer.

The VSMLIST service routine updates the work area and places the requested information in the data area located at the end of the work area. If the macro was executed successfully and all of the requested information fit into the data area, the VSMLIST service routine returns to the caller with a return code of 0 in the first four bytes of the work area. If the macro was executed successfully, but all of the requested information could not fit into the data area, the service routine returns to the caller with a return code of 4 in the first four bytes of the work area. In this case, the caller can reissue the macro as many times as necessary to obtain all of the information.

For multiple invocations of VSMLIST, the service routine continues supplying the information, starting where it left off on the previous invocation, provided the work area is not changed. However, multiple invocations do not provide cumulative results. For each invocation of a set of multiple invocations for a specific request:

- The count fields are relative to the current invocation of the macro (for example, the number-of-subpools field contains the number of subpool descriptors in the current invocation only).
- The output in the data area describes the current invocation only.

You can avoid multiple invocations by enlarging the work area to hold all of the information. If you do enlarge the work area, be sure to set the first four bytes of the work area (the return code area) to zero before reissuing the macro.

Table 33. Description of VSMLIST Work Area

Bytes	Field name	Description
0-3	Return code	This field contains the return code from the previous invocation of the VSMLIST macro. You must set this field to zero before the first invocation of the VSMLIST macro for a single request.
4-7	Address of data area	The data area is located at the end of the work area and contains the information that you requested.
8-11	Length of the data area	The data area varies in length and is limited in size by the length of the work area that you specified as a parameter of the VSMLIST macro.
12-15		Reserved for IBM use
32-x		Reserved for IBM use

Table 33. Description of VSMLIST Work Area (continued)

Bytes	Field name	Description
x-y	Data area	This area contains the actual output of the VSMLIST macro. The area varies in size and is limited by the length of the work area specified as a parameter of the macro.

The information returned in the data area depends on the parameters specified on the macro invocation. You can use the VSMLIST macro to obtain information about the following types of storage:

- Allocated
- Free
- Unallocated

Except for subpool 245, an allocated block of storage is a multiple of 4K, some of which has been allocated through a GETMAIN or STORAGE macro. Free space within that block is the area that has not been allocated by a GETMAIN or STORAGE macro. An unallocated block of storage is some multiple of 4K, none of which has been allocated by a GETMAIN or STORAGE macro.

VSMLIST reports all SQA pages not allocated to subpools 226, 239, 247, and 248 as allocated to subpool 245. These pages of subpool 245 may not have been allocated by a GETMAIN or STORAGE macro.

The format of the information returned in the data area for each of these three types of requests follows.

Allocated storage information

You can request allocated storage information by coding the SPACE=ALLOC parameter of the VSMLIST macro. The format of the output varies according to what you specify for the SP parameter.

If you specify SP=SQA, SP=CSA, or SP=LSQA, the output consists of the allocated storage information for the subpools in the specified area. The subpools listed in each of these areas are:

- SQA: 226, 239, 245, 247, 248
- CSA: 227, 228, 231, 241
- LSQA: 205, 215, 225, 255

Figure 37 on page 235 shows the output the system produces when you issue the VSMLIST macro with SP=CSA or SP=SQA specified.

If you specify SP=PVT, the output consists of the allocated storage information for subpools in the private area according to the owning TCB. These subpools are 0-127, 129-132, 229, 230, 236, 237, 244, 249, 251, and 252.

Figure 40 on page 237 shows the format of the allocated storage information for the private area.

If you specify a subpool list, the output consists of the allocated storage information for each of the subpools in the list. Figure 41 on page 238 shows the format of the allocated storage information for a subpool list request.

See the following figures for the format that the system uses to provide allocated storage information.

- For the format of the work area that VSMLIST produces when you specify SP=LSQA, SP=CSA, or SP=SQA, see Figure 37 on page 235
- For the format of the subpool descriptor, see Figure 38 on page 236
- For the format of the allocated block descriptor, see Figure 39 on page 236
- For the format of the allocated storage information for the private area, see Figure 40 on page 237
- For the format of the allocated storage information for a subpool list, see Figure 41 on page 238
- For the format of free space information ,see Figure 42 on page 238
- For the format of the free space descriptor, see Figure 43 on page 238
- For the format of unallocated storage information for CSA and PVT subpools, see Figure 44 on page 239
- For the format of the region descriptor, see Figure 45 on page 240
- For the format of the unallocated block descriptor, see Figure 46 on page 240

Number of subpools (4-byte field)	
First subpool descriptor	First subpool in area
Number of allocated blocks in first subpool	
Allocated block descriptor for first block	
Free space information, if SPACE=FREE is specified	
Allocated block descriptor for second block	
Free space information, if SPACE=FREE is specified	
:	
Allocated block descriptor for last block	
Free space information, if SPACE=FREE is specified	
Second subpool descriptor	Second subpool in area
Number of allocated blocks in second subpool	
Allocated block descriptor for first block	
Free space information, if SPACE=FREE is specified	
Allocated block descriptor for second block	
Free space information, if SPACE=FREE is specified	
:	
Allocated block descriptor for last block	
Free space information, if SPACE=FREE is specified	
Last subpool descriptor	Last subpool in area
Number of allocated blocks in last subpool	
Allocated block descriptor for first block	
Free space information, if SPACE=FREE is specified	
Allocated block descriptor for second block	
Free space information, if SPACE=FREE is specified	
:	
Allocated block descriptor for last block	
Free space information, if SPACE=FREE is specified	

This figure describes the work area that VSMLIST produces when you use SP=LSQA, SP=CSA, or SP=SQA. If you also specify SPACE=FREE, each allocated block descriptor is followed by free space information. If you specify SPACE=ALLOC, VSMLIST omits free space information.

Figure 37. Allocated Storage Information for Subpools in a Specified Area

Byte **Content**

0 X‘00’ to identify a subpool descriptor

1 Length of subpool descriptor

2 Subpool ID

3 Miscellaneous flags and storage key as follows:

- **Bit** **Meaning When Set**
- **0-3** Storage key
- **4** The TCB with which this descriptor is associated owns the storage described by this descriptor

  This is meaningful for private area storage only.
- **5** The storage described by this descriptor is shared. This is meaningful for private area storage only.
- **6-7** Reserved

4-7 Owning TCB address (if PVT subpool), otherwise zero.

Figure 38. Format of Subpool Descriptor

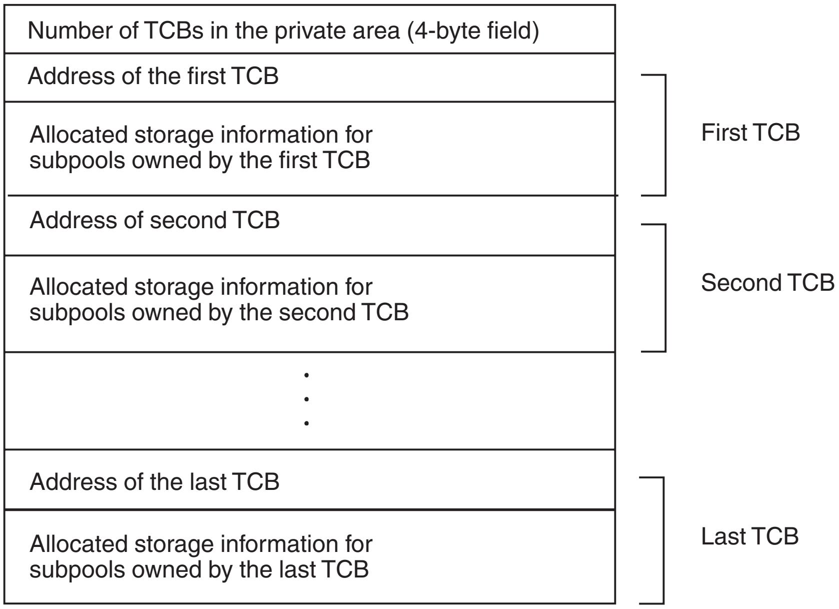
Byte **Content**

0-3 The virtual address of the allocated block

- **Bit** **Meaning When Set**
- **0** The caller specified the REAL31 option and this allocated block could be backed in central storage above (bit 0=1) or below (bit 0=0) 16 megabytes when the storage is fixed.
- **30** The caller specified the REALPGFMSZ option and this storage can be backed in central storage by 1 megabyte pages (bit 30=1) or by 4K pages (bit 30=0) when the storage is fixed.
- **31** The caller specified the REAL64 option and this allocated block could be backed in central storage above (bit 31=1) or below (bit 31=0) 2 gigabytes when the storage is fixed.

4-7 The length of the allocated block

Figure 39. Format of Allocated Block Descriptor


Number of TCBs in the private area (4-byte field)
Address of the first TCB
Allocated storage information for subpools owned by the first TCB
First TCB
Address of second TCB
Allocated storage information for subpools owned by the second TCB
Second TCB
. . .
Address of the last TCB
Allocated storage information for subpools owned by the last TCB
Last TCB

This figure describes the work area that VSMLIST produces when you use SP=PVT.

Note:

1. The number of subpools in the private area is a 31-bit number.
2. The address of the TCB is contained in a fullword.
3. Figure 37 on page 235 shows the format of the allocated storage information for the subpools owned by each TCB. The storage information begins at offset 0, "number of subpools" in that figure.
4. If you specify SPACE=FREE, free space information follows each allocated block descriptor. Figure 42 on page 238 describes the format of the free space information.

Figure 40. Allocated Storage Information for the Private Area

Number of SQA subpools (4-byte field)
Number of CSA subpools (4-byte field)
Number of LSQA subpools (4-byte field)
Number of TCBs
If the information of SQA subpools is not zero, the information for the SQA as described in "Allocated Storage Information for Subpools in a Specified Area" figure, beginning at offset 4
If the information of CSA subpools is not zero, the information for the CSA as described in "Allocated Storage Information for Subpools in a Specified Area" figure, beginning at offset 4
If the information of LSQA subpools is not zero, the information for the LSQA as described in "Allocated Storage Information for Subpools in a Specified Area" figure, beginning at offset 4
If the number of TCBs is not zero, the information for the private area as described in "Allocated Storage Information for Subpools in a Specified Area" figure, beginning at offset 0

This information is always present but could be zero.

This figure describes the work area that VSMLIST produces when you use SP=(subpool1,subpool2,subpool3,...).

Note: The number of subpools is a 31-bit number.

Figure 41. Allocated Storage Information for a Subpool List

Free space

The following blocks contain free space information.

Number of free space descriptors that follow (4-byte field)
First free space descriptor
Second free space descriptor
. . .
Last free space descriptor

This information is always present but may be zero.

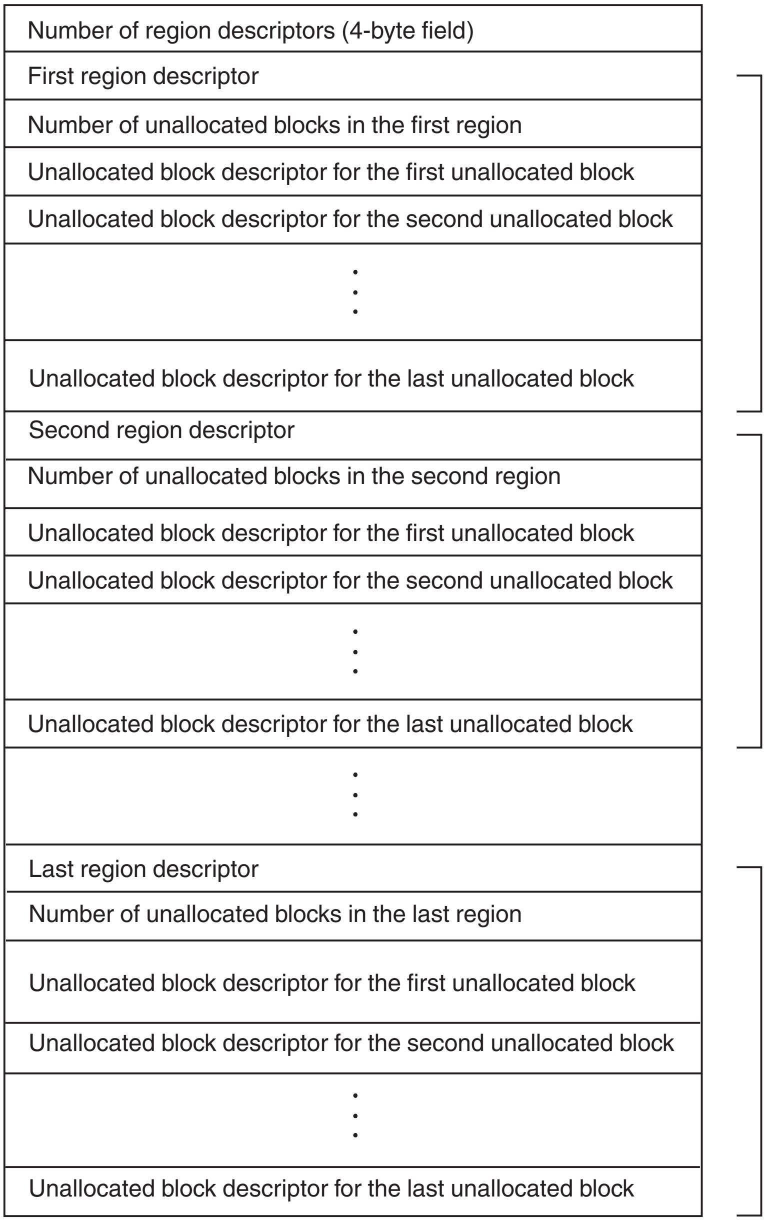
Figure 42. Format of Free Space Information

Byte	Content
0-3	The virtual address of the start of the free space
4-7	The length of the free space

Figure 43. Format of Free Space Descriptor

Unallocated storage information

You can request information about unallocated storage by specifying the SPACE=UNALLOC parameter of the VSMLIST macro. You can obtain this information for CSA and private area subpools only, by specifying SP=CSA or SP=PVT. Figure 44 shows the format of the output for a SPACE=UNALLOC request for CSA or PVT subpools.



Number of region descriptors (4-byte field)

First region descriptor

Number of unallocated blocks in the first region

Unallocated block descriptor for the first unallocated block

Unallocated block descriptor for the second unallocated block

.
.
.

Unallocated block descriptor for the last unallocated block

Second region descriptor

Number of unallocated blocks in the second region

Unallocated block descriptor for the first unallocated block

Unallocated block descriptor for the second unallocated block

.
.
.

Unallocated block descriptor for the last unallocated block

.
.
.

Last region descriptor

Number of unallocated blocks in the last region

Unallocated block descriptor for the first unallocated block

Unallocated block descriptor for the second unallocated block

.
.
.

Unallocated block descriptor for the last unallocated block

First region in specified area
Second region in specified area
Last region in specified area

Note:

1. The number of region descriptors and the number of unallocated blocks in each region are given as 31- bit numbers.
2. Figure 45 on page 240 shows the format of a region descriptor.
3. Figure 46 on page 240 shows the format of an unallocated block descriptor.

Figure 44. Unallocated Storage Information for CSA and PVT Subpools

Byte	Content
0-3	The virtual address of the region (CSA, ECSA, RCT area, V=V area, extended V=V area, or V=R area)
4-7	The length of the region

Figure 45. Format of Region Descriptor

Byte	Content
0-3	The virtual address of the unallocated block
4-7	The length of the unallocated block

Figure 46. Format of Unallocated Block Descriptor

Using IARQD — The page status interface routine

If you are writing a program that dumps virtual storage and you need to determine where the virtual storage pages of an address space or of a data space reside, use the Page Status Interface Routine (IARQD).

Before using IARQD, you must understand the types of pages managed in virtual storage and how these pages might be managed through the paging hierarchy. For additional information, see:

- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Extended Addressability Guide*

When you invoke IARQD, a search begins for a page location and status. This search begins at an address that you provide, and continues for the number of pages you specify unless you reach a change in the page status or the end of storage first.

The requirements for the caller are:

Environmental factor	Requirement
Minimum authorization:	Supervisor state with PSW key 0
Dispatchable unit mode:	Task or SRB
Cross memory mode:	PASN=HASN=SASN
AMODE:	31-bit
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No SPIN locks
Control parameters:	The parameter list must be in non-pageable storage in the caller's primary address space.

There are 4 steps involved in using this interface:

1. Decide which of the 2 entry points you want to use, IARQDUMP or IARQDSPD.
2. Obtain storage and load register 1 with the address of the parameter list
3. Use NUCLKUP to find the address of the entry point you want to use
4. Invoke the entry point and evaluate the returned information
   - Return codes

- Reason codes (possibly)
- Delimiter page

Decide which entry point you want to use

Entry point **IARQDUMP** determines the location and status of a page in the storage hierarchy of a page in an ***address space***. See Figure 1 for return codes and reason codes.

Entry point **IARQDSPD** determines the location and status of a page in the storage hierarchy of a page in a ***data space***. See Figure 1 for return codes and reason codes.

Obtain storage and load register 1

IARQDUMP

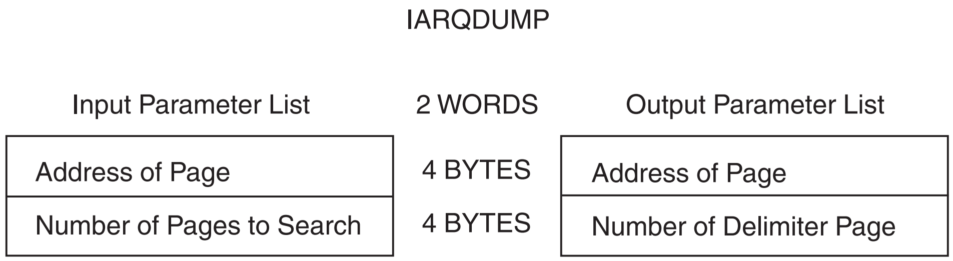
R1 contains the address of a 2-word parameter list

The contents of the input parameter list is mapped as follows:

- The first word contains the virtual address of any byte in the page to start looking at
- The second word contains the maximum number of pages to process.

The content of the output parameter list for **IARQDUMP** is mapped as follows:

- The first word remains unchanged
- The second word contains the address of the **delimiter page**.



IARQDUMP

Input Parameter List	2 WORDS	Output Parameter List
Address of Page	4 BYTES	Address of Page
Number of Pages to Search	4 BYTES	Number of Delimiter Page

IARQDSPD

R1 contains the address of a 4-word parameter list

The input parameter list is:

- The first word contains the virtual address of any byte in the page to start looking at
- The second word contains the maximum number of pages to process
- The third and fourth words contain the 8-byte STOKEN for the data space to be examined.

The output parameter list for **IARQDSPD** is:

- The first word remains unchanged
- The second word contains the address of the **delimiter page**
- The third and fourth words remain unchanged.

IARQDSPD

Input Parameter List	4 WORDS	Output Parameter List
Address of Page	4 BYTES	Address of Page
Number of Pages to Search	4 BYTES	Number of Delimiter Page
Data space STOKEN	8 BYTES	Data space STOKEN

The delimiter page

The **delimiter page** is the next page after your search has stopped. It may or may not have a different page status than the last page searched.

- If the search goes on for the maximum number of pages (specified in the second word of the parameter list), and still does not find a different page status, then the address of the next page after the last page examined is returned as the delimiter page. This page may or may not have a different page status than the last page examined.
- If the search finds a different page status before reaching the maximum number of pages to be searched, the search is complete and the address of the next page after the last page examined is returned as the delimiter page. This page does have a different page status (that is why the search ended).
- If the search reaches the end of storage before finding a different page status, the delimiter page will contain the address X'7FFFFFFF'.
- The search may be terminated before the maximum number of pages have been searched, or before a page of a different status has been found and before the end of storage has been reached. In that case, all pages up to the delimiter page have the same status and the status of the delimiter page may or may not be the same as the previous page.

  This might occur, for example, when the page table for the delimiter page has been paged out.

Use NUCLKUP to find the address of the entry point you want to use

NUCLKUP will put the address of the entry point in Register 0.

For more information on the NUCLKUP macro, see *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

Example

Operation: Place the address and AMODE of entry point IARQDUMP in register 0.

```
NUCLKUP BYNAME,NAME='IARQDUMP',ADDR=(0)
```

Input register information

Before calling either the IARQDUMP or IARQDSPD entry point, the caller must make sure that the following general purpose registers (GPRs) contain the specified information:

Register	**Contents**
0	Used as a work register

1 The address of a non-pageable parameter list

2-12 Used as work registers

13 The address of a standard save area

14 The return address

15 The entry address

Before calling either the IARQDUMP or IARQDSPD entry point, the caller does not have to place any information into the access registers.

Invoke the entry point

To invoke the entry point you have chosen, use the following instructions:

- LR 15,0
- BALR 14,15

The output returns are:

- Return code
- Reason code (possibly)
- Delimiter page

When control returns to the caller, the GPRs contain:

Register
Contents

0 The reason code if R15 is 08.

1 The address of the parameter list

2-14 Unchanged

15 The return code

When control returns to the caller, the ARs are unchanged.

When the IARQDxx entry point returns control to your program, GPR 15 contains the return code. If the return code is 8, GPR0 contains the reason code.

Table 34. Return and Reason Codes for IARQD

Return Code	Reason Code	Meaning
00	none	A copy of the page cannot be found in virtual storage. (that is: real storage, expanded storage, or auxiliary storage) This could happen because: • The page was not obtained through the GETMAIN or STORAGE macro • The page was just obtained and has no data in it yet • The page cannot be read into real storage due to an I/O error
04	none	All copies of the page are in virtual storage at the same level as what is on permanent storage. This return code is only possible for pages that have been mapped through the DIV (data-in-virtual) macro.
08	xx0000xx	The page is in central storage. At least one copy of the page has been found in virtual storage. For data-in-virtual (DIV) pages, the copy of the page is at a later level than the copy on permanent storage.

Table 34. Return and Reason Codes for IARQD (continued)

Return Code	Reason Code	Meaning
08	xx0004xx	The page is on a synchronous medium, such as in expanded storage At least one copy of the page has been found in virtual storage. For data-in-virtual (DIV) pages, the copy of the page is at a later level than the copy on permanent storage.
08	xx0008xx	The page is on an asynchronous medium At least one copy of the page has been found in virtual storage. For data-in-virtual (DIV) pages, the copy of the page is at a later level than the copy on permanent storage.
0C		The status of the page is not determined

Chapter 11. Accessing the scheduler work area

When the system interprets a job's JCL statements, it obtains information about the job's data sets. It stores this information in the scheduler work area (SWA). When the job runs, the system develops additional information about the job's data sets, which it also stores in the SWA. Some of this information is in the following SWA blocks:

- The job file control block (JFCB)
- The job file control block extension (JFCBX)

Your program can use the SWAREQ macro and the IEFQMREQ macro to read from a block, write into a block, or obtain the location of a block. The only SWA blocks that you can access are the ones associated with your job.

A SWA block can be accessed using one of the following:

- IEFQMREQ reads the SWA information into a buffer that you provide, or writes information from your buffer into the SWA.
- SWAREQ, instead of actually writing or reading information, only tells you the location of the SWA block that you are interested in. Once you know the location, you can examine or change the block directly.

Use IEFQMREQ to access the SWA in programs that have a 24-bit addressing mode and SWA located above 16 megabytes. See *z/OS JES2 Initialization and Tuning Guide* or *z/OS JES3 Initialization and Tuning Guide* for information on placing SWA above or below 16 megabytes.

Using the IEFQMREQ and the SWAREQ macros

To use these macros, a program must be in supervisor state, in task mode, and not in cross memory mode. However, when you are using SWAREQ to perform a Read Locate, you can override these restrictions by specifying the UNAUTH=YES parameter. Overriding these restrictions by specifying the UNAUTH=YES parameter does not mean that the SWA blocks for another address space can be accessed, but that the service can run with the primary address space even if it is not the same as the home address space.

To use these macros, you must provide information to one of two external parameter areas (EPAs). When invoking SWAREQ, use the EPAL. When invoking IEFQMREQ, use the EPAM. This information uses the term EPA when generically referring to either of these areas.

You must provide a function code and a pointer to an EPA. The function code specifies the service that the macro is to perform. The EPA is where you store input data to the macro and where the macro returns output data to you. The input data in the EPA depends on the function code that you specify. The data in the EPA can be:

- The buffer address.
- The token that represents the SWA block. These tokens are called SVAs.
- The pointer (block pointer) to the SWA block being accessed.
- The length (block length) of the block being accessed.

- An ID field (block ID) that represents the type of SWA block. Use this block ID to compare against the block ID in a SWA block returned from a read function. If the comparison is not equal, then the returned block is not the type of SWA block that you requested. Block ids are mapped by the IEFQMIDS mapping macro.

One of the items that you must store in the EPA before invoking the macro is the token that identifies the SWA block that you want to access. You can obtain these tokens (called SVAs) from the following fields:

SWA block to be accessed	Field that contains the token	Mapping macro for field that contains the token
JFCB (job file control block)	TIOEJFCB in the TIOT entry	IEFTIOT1
JFCBX (job file control block extension)	JFCBEXAD in the JFCB	IEFJFCBN

The SWAREQ macro

SWAREQ, instead of actually writing or reading information, only tells you the location of the SWA block that you are interested in. Once you know the location, you can read or write information yourself. By specifying a function code when you invoke the macro, you can request the following SWAREQ services.

- **Read Locate —** Returns the address of the block that you specify. It does not read any data from the specified block into your buffer. Your program does the actual reading by coding techniques such as MVC instructions.
- **Write Locate —** When you use regular coding techniques to write data from your buffer into the data area of a SWA block, the system does not know that the block has been written into. To allow the system to set up the control fields that are necessary to integrate the SWA block into the system, use write locate to inform the system that a write has taken place.

How to invoke SWAREQ

As parameters of the SWAREQ macro, you specify the function code and the pointer to the EPAL. The EPAL input data and the EPAL output data for each function code is summarized in the following block:

SWAREQ Function	EPAL Size	EPAL Input Fields	EPAL Output Fields
Read Locate	16 and 28	SVA	Block pointer, block length, block ID
Write Locate	16	SVA, block pointer, block ID	None

When you write a program that invokes SWAREQ, you must provide the field definitions in the EPA. You might also need to provide the SWA block definitions. When you assemble the program, the following mapping macros provide the definitions that you need:

- IEFZB505 — EPAL data area
- IEFJESCT — JES control block
- CVT — communications vector table
- IEFQMIDS — SWA block ID definitions

When you specify UNAUTH=YES, you must observe the following rules:

- Your code must be running in 31–bit mode.
- Your function code must specify Read Locate.
- The EPAL that you provide must be an extended EPAL. To provide an extended EPAL, use the LOCEPAX=YES parameter of the IEFZB505 mapping macro. **Note** that the longer 28 byte EPAL is generated with LOCEPAX=YES.
- If the job for which you are invoking SWAREQ is not the current job, you must pass the queue manager parameter area (QMPA) address that is associated with the job you are interested in. In this case, obtain the QMPA address from the active JSCB of whatever job you are interested in, and place this address in the EPAL that SWAREQ refers to by its input parameter. Data is retrieved from the current primary address space.

SWAREQ summary

To issue a locate mode request, take the following steps:

1. Build an EPAL (mapped by macro IEFZB505).
2. Issue the SWAREQ macro, specifying the address of the EPAL pointer and the required function code.

Example of using SWAREQ

The following program locates the JFCB block in the scheduler work area. After the program obtains the location of the block, it can store new information in the block or it can move information from the block into another area. The example assumes that general purpose register (GPR) 6 points to the TIOT:

- **UNAUTH=NO specified or defaulted to:**

```
         LA     5,EPA                  GET ADDRESS OF THE EPA
         ST     5,SWEPAPTR             INITIALIZE EPA POINTER
         USING  ZB505,5                ESTABLISH ADDRESSABILITY TO EPA
         XC     SWAEPA,SWAEPA          INITIALIZE THE EPA
         USING  TIOT1,6                ESTABLISH ADDRESSABILITY TO TIOT
         MVC    SWVA,TIOEJFCB          MV SVA OF JFCB INTO EPA
         SWAREQ FCODE=RL,EPA=SWEPAPTR,MF=(E,SWAPARMS)  LOCATE THE JFCB
         L      7,SWBLKPTR             SET THE POINTER TO THE JFCB
         USING  INFMJFCB,7             ESTABLISH ADDRESSABILITY TO JFCB
*
SWEPAPTR DS    F
EPA      DS    CL16
SWAPARMS SWAREQ MF=L
         CVT DSECT=YES
         IEFJESCT
         IEFZB505
         TIOT DSECT
         IEFTIOT1
         IEFJFCBN
```

- **UNAUTH=YES specified:**

```
         LA     5,EPA                  GET ADDRESS OF THE EPA
              ST     5,SWEPAPTR             INITIALIZE EPA POINTER
              USING  ZB505,5                ESTABLISH ADDRESSABILITY TO EPA
              XC     SWAEPAX,SWAEPAX        INITIALIZE THE EPA
              USING  TIOT1,6                ESTABLISH ADDRESSABILITY TO TIOT
              MVC    SWVA,TIOEJFCB          MV SVA OF JFCB INTO EPA
              SWAREQ FCODE=RL,EPA=SWEPAPTR,MF=(E,SWAPARMS)  LOCATE THE JFCB
              L      7,SWBLKPTR             SET THE POINTER TO THE JFCB
              USING  INFMJFCB,7             ESTABLISH ADDRESSABILITY TO JFCB
     *
     SWEPAPTR DS    F
     EPA      DS    CL28
     SWAPARMS SWAREQ MF=L
              CVT DSECT=YES
              IEFJESCT
```

```
IEFZB505  LOCEPAX=YES
TIOT DSECT
IEFTIOT1
IEFJFCBN
```

Return codes and reason codes from SWAREQ

UNAUTH=YES: If you specify UNAUTH=YES, SWAREQ cannot abend. It always returns to the program that invoked it. Check the return code in GPR 15. If the return code is 0, the service is successful. Otherwise, the service failed and the non-zero return code in GPR 15 is also the reason code associated with the failure.

UNAUTH=NO: If you specify UNAUTH=NO or omit UNAUTH, the service can abend if an error occurs or if your program is holding a lock. The return, reason, and abend codes for UNAUTH=NO are as follows:

When control returns after invoking SWAREQ, check the return code in GPR 15. If the return code is 0, the service is successful. Otherwise, the service failed, and the non-zero return code in GPR 15 is also the reason code associated with the failure. There is only one reason code: reason code hexadecimal 38, which means that the system could not obtain the storage necessary to carry out the request.

When control does not return from SWAREQ, an abend occurred. To interpret the abend dump, use the contents of GPRs 1 and 15. GPR 0 contains the address of an area that contains diagnostic information. This area is further described in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*. GPR 1 contains abend code 0B0. GPR 15 has the reason code associated with the abend.

The IEFQMREQ macro

IEFQMREQ reads the SWA information into a buffer that you provide, or writes information from your buffer into the SWA. By specifying a function code when you invoke the macro, you can request the following IEFQMREQ services. Use the symbolic function codes that are in the QMPA mapping macro:

- **Read —** Reads the data area of a specified block into your buffer. IEFQMREQ reads 176 bytes of data from SWA into your buffer. If the requested data is less than 176 bytes in length, IEFQMREQ pads the data with zeroes so the amount read into your buffer is 176 bytes.
- **Write —** Writes information from your buffer into the block that you specify. Only the data area is written, not the prefix area.

These are the only general-use programming interface function codes.

How to invoke IEFQMREQ

The IEFQMREQ macro does not have any parameters. Before you invoke the macro you must store input data for the macro in the QMPA and the external parameter area (EPAM). The input that you store in the QMPA is:

- **The function code —** In the field **QMPOP**, the function code specifies the service to be performed.
- **The EPAM address —** The EPAM address, which can be in either of two QMPA fields, locates the EPAM. Fill in the **QMPACL** field if you are specifying a three-byte address and the **QMPACLX** field if you are specifying a four-byte address.
- **A 4-byte EPAM address indicator —** Set this indicator in the field **QMPACLEX** if you are using a four-byte EPAM address.

- **The extended EPAM indicator —** Some services let you specify the size of the EPAM as 8 or 16 bytes. Set this indicator in the field **QMPAX** if you are using the 16-byte EPAM size.
- **The number of EPAMs —** This is the number of times that the function is to be performed, and the number of EPAM blocks that you are passing. Specify this number in the **QMPNC** field. For example, when you read three different SWA blocks into three different buffers in a single invocation of IEFQMREQ, the number that you specify in this field is 3.

If you want the function to be performed more than once, supply more than one EPAM. For example, you can read three different SWA blocks into three different buffers in a single invocation of IEFQMREQ. If you supply more than one EPAM, you must arrange them contiguously in storage. When you invoke the macro, GPR 1 must point to the QMPA. The EPAM input data and the EPAM output data for each IEFQMREQ function code are summarized in the following block:

IEFQMREQ Function	EPAM Size	EPAM Input Fields	EPAM Output Fields
Read	8 or 16	SVA, buffer address	Block ID
Write	8 or 16	SVA, buffer address, block ID	None

When you write a program that invokes IEFQMREQ, in addition to supplying input data in fields that are in the QMPA and the EPAM, you must supply SWA block definitions as input to the macro. When you assemble the program, the assembler needs definitions for the CVT and the JESCT. The format of the input data is defined in the following mapping macros:

- CVT — communications vector table
- IEFJESCT — JES control block
- IEFQMIDS — SWA block id definitions
- IEFQMNGR — QMPA data area
- IEFZB506 — EPAM data area

Example of using IEFQMREQ

The following program copies the JFCB from the scheduler work area into a buffer that the program provides. The example assumes GPR 6 points to the TIOT:

```
         LA     5,EPA                  GET ADDRESS OF THE EPA
         USING  SWAMMEPA,5             ESTABLISH EPA ADDRESSABILITY
         LA     1,QMPA                 GET ADDRESS OF THE QMPA
         USING  IOPARAMS,1             ESTABLISH QMPA ADDRESSABILITY
         XC     IOPARAMS(36),IOPARAMS  INITIALIZE THE QMPA
         MVI    QMPOP,QMREAD           INDICATE READ FUNCTION
         MVI    QMPCL,1                INDICATE 1 EPA IS BEING PASSED
         STCM   5,7,QMPACL             PUT 3-BYTE EPA ADDRESS IN QMPA
         XC     SWAMMEPA,SWAMMEPA      INITIALIZE THE EPA
         USING  TIOT1,6                ESTABLISH ADDRESSABILITY TO TIOT
         MVC    SWROWVA,TIOEJFCB       SVA OF JFCB MOVED TO EPA
         LA     8,JFCBCOPY             SET THE POINTER TO THE JFCB
         ST     8,SWBUFPTR             SET BUFFER POINTER IN EPA
         IEFQMREQ                      COPY SWA BLOCK TO THE BUFFER
         USING  INFMJFCB,8             ESTABLISH ADDRESSABILITY TO JFCB
*
JFCBCOPY DS    CL176                   BUFFER TO READ THE JFCB INTO
EPA      DS    CL8
QMPA     DS    CL36
         CVT DSECT=YES
         IEFJESCT
         IEFZB506
         IEFQMNGR
TIOT     DSECT
         IEFTIOT1
         IEFJFCBN
```

Return codes and reason codes from IEFQMREQ

When control returns after invoking IEFQMREQ, check the return code in GPR 15. If the return code is 0, the service is successful. Otherwise, the service failed, and the non-zero return code in GPR 15 is also the reason code associated with the failure. There is only one reason code: reason code hexadecimal 38, which means that the system could not obtain the storage necessary to carry out the request.

When control does not return from IEFQMREQ, an abend occurred. To interpret the abend dump, use the contents of GPRs 0, 1 and 15. GPR 0 contains the address of an area that contains diagnostic information, described further in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*. GPR 1 contains abend code 0B0. GPR 15 has the reason code associated with the abend.

Chapter 12. The virtual lookaside facility (VLF)

Virtual lookaside facility (VLF) is a set of services that can improve the performance and response time of applications that must retrieve a set of data for many users. VLF creates and manages a data space to store an application's most frequently used data. When the application makes a request for data, VLF checks its data space to see if the data is there. If the data is present, VLF can rapidly retrieve it without requesting I/O to DASD.

To take advantage of VLF, an application must identify the data it needs to perform its task. The data is known as a data object. Data objects should be small to moderate in size, named according to the VLF naming convention discussed later in this information, and associated with an installation-defined class of data objects.

VLF is intended for use with major applications. Because VLF runs as a started task that the operator can stop or cancel, it cannot take the place of any existing means of accessing data on DASD. Any application that uses VLF must also be able to run without it.

An installation can realize immediate benefits from VLF by using some of the products that use VLF. See *z/OS MVS Initialization and Tuning Guide* for information about how to achieve these benefits.

Deciding when to use VLF

Before creating a new application or recoding an existing one to take advantage of VLF, you must answer a few questions about the data objects your application will use.

- **What kind of data objects does your application work with?**

  VLF works best with two kinds of data: (1) data objects that are members of partitioned data sets, located through a partitioned data set (PDS) concatenation, or (2) data objects that, while not PDS members, could be easily described as a collection of named objects that are repeatedly retrieved by many users.

  If neither description fits your data objects, it is likely that you would not obtain any performance benefit from VLF. An application that retrieves named shared objects might save development cost by using VLF, but there are storage overhead costs associated with using VLF, and it is best used when you expect performance benefit.
- **Is your data accessed frequently enough so that much of it is likely to remain in processor storage?**

  Like data in private storage, data stored through VLF is subject to page stealing. Thus, appropriate data for VLF is data shared by many users, so that the combined reference rate of all users is likely to keep the data in processor storage. That is, VLF works best when a significant portion of the data is likely to remain in processor storage and not be paged out to auxiliary storage.
- **How large are your data objects?**

  VLF works best with relatively small objects because less virtual storage is expended to reduce the number of I/O operations. Very large objects, if they are not used frequently enough to remain in central or expanded storage, might take longer to retrieve through VLF than through traditional I/O from DASD.

- **How long do you expect users to actively retrieve objects through your application?**

  VLF processing adds overhead to the initial retrieval of an object from DASD, and your application must identify users to VLF. Some applications might require more identifications than others. If many retrieves are not likely for each identification, then the relative benefit of using VLF is less.

Other questions that are important to answer when deciding whether or not to use VLF are:

- **Is your system storage-constrained?**

  VLF is designed to improve performance by increasing the use of virtual storage to reduce the number of I/O operations. For a system that is already experiencing a central, expanded, or auxiliary storage constraint, this strategy is probably not a good choice.
- **Is user access to data controlled at the address space level or at the task level?**

  When your application identifies an end user to VLF, VLF returns a user token. Any task in the user address space can then use that token. Thus, VLF checks access to data objects at the address space level. You cannot use VLF to manage data that must be restricted to only certain tasks in an address space.
- **Is running in either supervisor state or with PSW key mask 0-7 a problem for your application?**

  This environmental restriction is essential to maintain system integrity. In addition, if your application obtains its VLF data objects from DASD, you must be sure that the program that reads in the object and creates the VLF object maintains the integrity of the data.
- **Does your application process the data obtained from DASD in a repetitive way before passing it to each user?**

  If it does, you can potentially obtain additional benefit from VLF by saving preprocessed objects in VLF storage rather than exact copies of the DASD objects. This technique would require some recoding for an existing application, but it could provide a significant benefit if the amount of common repetitive processing is large.

Planning to use VLF

To activate VLF for your application, your installation must do two things:

1. Start VLF. See *z/OS MVS System Commands* for information on how to use the START command to start VLF.
2. Update COFVLFxx to include the VLF class associated with the application or product.

   See *z/OS MVS Initialization and Tuning Reference* for detailed information about creating COFVLFxx.

An installation may need to add the VLFNOTE command to the TSO/E authorized command table. To update the TSO/E authorized command table, define an entry in the SYS1.PARMLIB member IKJTSOxx. For information on IKJTSOxx, see *z/OS MVS Initialization and Tuning Reference*. For information on the VLFNOTE command, see *z/OS TSO/E Command Reference*.

As you decide how to use VLF, you must also consider whether the end user of your application needs to know about VLF. The end user, perhaps a TSO/E user, will define the object required; your application will retrieve that object through VLF and then present it to the end user.

In an installation that has a single system (which can be a tightly-coupled multiprocessing system), the end user normally needs no awareness of VLF; the only effect is improved performance and response time.

In an installation with multi-system shared DASD, the end user needs some awareness of VLF and, on occasion, might need to communicate directly with VLF. If you use VLF in such an environment, you need to provide education for end users on how VLF affects their work and what kind of changes VLF might create. See "Cross-system sharing" on page 254.

Within your application, you must give particular consideration to:

- The data objects VLF is to manage, particularly if you expect users to have private data sets or if the data sets are shared across systems.
- Protecting the integrity of the data.
- Recovery for your application.

"Using the VLF macros" on page 257 describes the macros you use to request VLF services.

Data objects and classes

When developing an application that uses VLF, you must use the VLF naming convention when naming VLF data objects. Data objects managed by VLF are structured by your installation into groups called classes. Each class can represent data managed by a different application. The IBM products that use VLF have specific class names defined in the COFVLFxx member of SYS1.SAMPLIB. Installation-written applications cannot use these names for their own classes of data.

Thus, a **class** is a group of related objects made available to users through an application. Note that a named object may belong to one class only. This restriction means that an object may be retrieved using VLF only by the application that manages the class the object belongs to.

VLF provides two different implementations of its services that are dependent on what class the object belongs to. A class that consists of objects that correspond to members of partitioned data sets is called a **PDS class**. Objects belonging to a PDS class are updated by VLF whenever the data object is changed on DASD. A class that consists of objects that do not correspond to members of partitioned data sets is called a **non-PDS class**. Objects belonging to a non-PDS class are not updated by VLF whenever the data object is changed on DASD. When designing your application, you may even choose a non-PDS implementation of VLF with data objects that belong to a partitioned data set.

VLF determines whether it is dealing with a PDS or non-PDS class from the class definition in the COFVLFxx parmlib member specified when VLF is started. In both the PDS and non-PDS class, VLF uses two levels of names to identify an object:

1. **Major name —** specifies a subgroup of objects within a class. The major name for a PDS class is created by combining the volume serial number and data set name.

   Within a class, each major name must be unique; the same major name, however, can exist under more than one class name.
2. **Minor name —** specifies a specific object within a major name. The minor name of a VLF data object belonging to a PDS class is the PDS member name

of the source data. Within a major name, each minor name must be unique; the same minor name, however, can exist under more than one major name.

Thus, for a given class with several major names, multiple objects might have the same minor name.

This naming structure parallels the existing structure used to access members of partitioned data sets. The major name corresponds to a combination of the volume serial number and the name of the partitioned data set; that is, it uniquely identifies a group of objects. The minor name corresponds to a PDS member name; it uniquely identifies a specific data object by name.

Locating a specific minor name within a VLF class is thus very much like the process of locating a specific member name within a set of concatenated partitioned data sets. VLF allows a unique major name search order for each end user. The order determines which major names are searched first for that user.

For a PDS class, VLF determines a user's search order by DDNAME specification and automatically removes the user's identification if the allocation changes in any way. In addition, VLF automatically purges objects from the class if they are updated in the corresponding partitioned data set on DASD. Note that automatic purging applies only to the local system, which can be a tightly-coupled system. See “Cross-system sharing” for information about sharing access to partitioned data sets across systems.

Private data sets

If you are planning to use a PDS class, you must consider whether individual users of your class are likely to have private data sets concatenated ahead of eligible data sets (data sets that contain members that VLF is to manage as data objects). VLF is less efficient when private data sets are present in a user's concatenation ahead of the eligible data sets.

Your application does not have to distinguish between private data sets and eligible data sets. VLF, however, does perform extra processing when private data sets precede eligible data sets. Specifically, if an end user has a private data set concatenated ahead of an eligible data set, the first attempt to retrieve a given object from VLF storage will not succeed.

If you have control over the search orders, VLF works most efficiently when private data sets (or ineligible major names for non-PDS classes) are either not allowed or follow the eligible names rather than precede them.

Cross-system sharing

To ensure that users have access to current data, VLF needs to be notified when changes are made to data that is shared across systems. For instance, if a user changes shared data but VLF on all the affected systems is not notified, VLF users on those systems would not have access to current data.

Under certain circumstances, VLF is automatically notified of changes to data it manages. At other times, someone, such as a user or an application programmer, needs to notify VLF that data has been changed. Table 35 on page 255 summarizes when notification to VLF is automatic. A more detailed explanation follows the figure.

Table 35. When VLF Notification is Automatic

Class of Data Object	Multiple Systems	VLF Notification
PDS	Sysplex	Automatic
PDS	No sysplex	Not automatic
Non-PDS	Sysplex	Not automatic
Non-PDS	No sysplex	Not automatic

When VLF notification is automatic

Notification to VLF is automatic when both of the following are true:

- VLF runs on a sysplex. Each VLF that is active on a sysplex automatically receives notification of changes to shared data. When one system in the sysplex changes data, each VLF that is active on the sysplex and that shares the changed data is automatically informed of the change. See *z/OS MVS Setting Up a Sysplex* for more information about setting up VLF to run on a sysplex.
- The changed data is in a PDS class. Additionally, the change to the PDS data must have been made by either:
  - A program that uses IBM data management services
  - A program that provides notification of the change to one of the systems by issuing the COFNOTIF macro.

When VLF notification is not automatic

If VLF is running on systems that are not part of a sysplex or the changed data is non-PDS data, notification to VLF is not automatic. Whoever controls the shared data must notify VLF on individual systems that the data has been changed.

How VLF is notified depends on who controls the data. For example, an end user in charge of the data would use the TSO/E VLFNOTE command to notify VLF. An application programmer might use the COFNOTIF macro in a program.

Which systems need to be notified depends on the type of data being changed:

- For PDS data, all the systems except the local system (the system on which the data change was made) need to be notified. VLF on the local system receives automatic notification as long as the change was made using IBM data management services. (If the change was made some other way, the local system needs to be notified.)
- For non-PDS data, VLF must be notified on all of the systems sharing the data, including the local system.

The types of changes that require VLF notification appear in the following lists.

For data in a PDS, notify VLF when:

- Adding a member to an eligible data set (a data set that is identified to VLF).
- Adding a member to a non-eligible data set when both of the following are true:
  - The data set containing the new member is in a user's SYSPROC concatenation ahead of an eligible data set.
  - The eligible data set has a member with the same name as the new member.
- Updating an existing member of an eligible data set.
- Deleting an eligible data set or member of an eligible data set.

For non-PDS data, notify VLF when:

- Adding a minor name to a major name.
- Updating a minor name associated with a major name.
- Deleting a minor name from a major name.

Note: If you plan to use the VLFNOTE command, be sure your installation defines VLFNOTE in the IKJTSOxx parmlib member. See *z/OS MVS Initialization and Tuning Reference* for a description of the IKJTSOxx parmlib member.

The VLFNOTE command can be issued from either:
- A TSO/E user ID on each system
- A batch job, with system affinity, on each system.

See *z/OS TSO/E Command Reference* for information about using the VLFNOTE command.

Data integrity

VLF constitutes a form of “shared” storage in that data placed into VLF as a result of one user's processing might later be returned to many other users. Given this fact, it is obvious that your application must include controls to protect the integrity of the data to be stored in VLF.

The VLF environment provides some of the required integrity. All VLF macros, with the exception of the object retrieval (COFRETRI) macro, require that the caller be running in supervisor state or with PSW key mask 0-7. Thus, VLF can guarantee that, once an object is created in VLF storage, it cannot be modified by unauthorized code. VLF cannot, however, guarantee that the data stored in VLF in the first place is correct; your application must ensure the integrity of the data to be stored in VLF.

The best way to provide this integrity is to design your application so that objects are created by authorized code using system key storage. See “Creating a VLF object” on page 261 for details.

Recovery

Your program must be able to handle cases when VLF is inactive. For example:
- VLF might be inactive when you issue a VLF macro.
- The operator might cancel or stop VLF, thus making it inactive, while VLF is processing your request and your application's address is swapped out.

If your program issues a VLF macro when VLF is inactive, the system returns a return code of X‘28’. Your program should then use an alternate method of retrieving data.

To avoid an abnormal termination that might occur if VLF becomes inactive while processing your request, your program should:
1. Establish a recovery routine before issuing any VLF macro.
2. Set flags before and after issuing any VLF macro. If an error causes your recovery routine to get control, the recovery routine can use the flags to determine whether the program issued a VLF macro, but had not yet regained control, when the problem occurred.
3. If the flags indicate the application was in a VLF macro when the failure occurred, the recovery routine can, in nearly all cases, assume that the failure occurred because VLF is inactive. The recovery routine can then take the same

action that your application does for a return code of X'28'. Generally, the recovery routine should not request a dump or write a record to the logrec data set for this failure. The routine can assume that the system has collected serviceability data for this unexpected failure.

Using the VLF macros

VLF provides seven basic functions. An application invokes each function by issuing a VLF macro. The functions and the corresponding macros are:

- Defining a class of VLF objects — COFDEFIN
- Identifying an end user to VLF — COFIDENT
- Retrieving a VLF object — COFRETRI
- Creating a VLF object — COFCREAT
- Notifying VLF of a change — COFNOTIF
- Removing a VLF end user — COFREMOV
- Purging a VLF class — COFPURGE

Use the information here to decide how to use VLF. *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* contains information about coding each macro, including the return and reason code combinations the macro issues to describe the processing performed.

As you design your application, consider these return and reason codes carefully. For example, VLF runs as a started task, and most installations would start VLF as part of the IPL process. It is possible, however, that VLF might not be active when your application issues VLF macros. Thus, your application must be able to run without VLF. Also, each VLF macro issues a return code of X'28' when VLF is not active; each time your application issues a VLF macro, it must check for this return code.

If your application must be able to run on multiple levels of MVS, note that all VLF macros issue return code X'28' on a level of MVS prior to MVS/SP3.

Figure 47 on page 258 shows, at a very high level, one possible way of using VLF to retrieve objects from a PDS class. The figure shows the sequence of steps an application might use to find and read a member without VLF, and the high-level process of retrieving an object for an end user with VLF. In the VLF process, each shaded box represents the same steps the application uses to find and read a member without VLF. Figure 47 on page 258 does not include all VLF functions or the detailed return code checking than an actual application would perform.

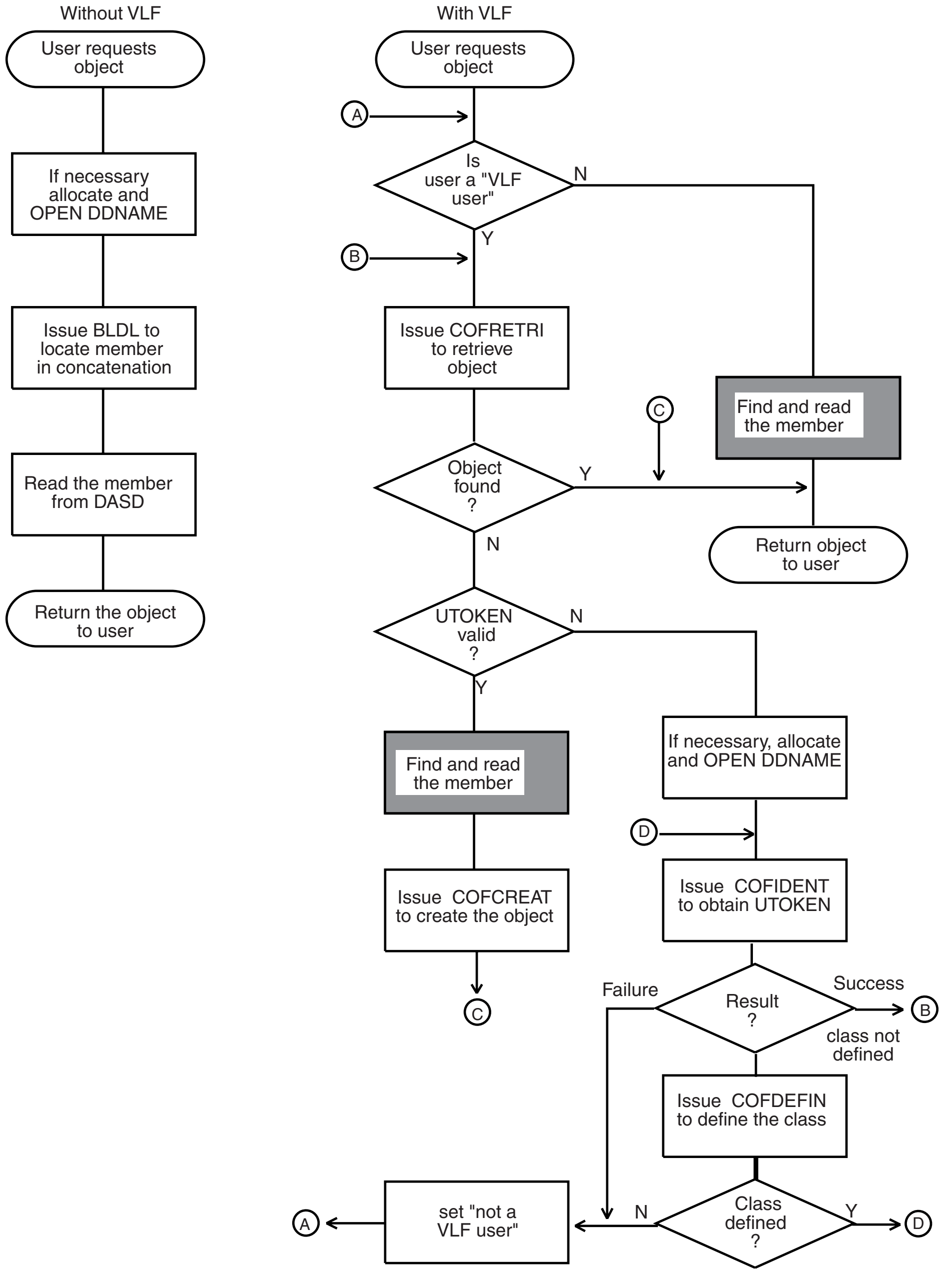

Without VLF
User requests object
If necessary allocate and OPEN DDNAME
Issue BLDL to locate member in concatenation
Read the member from DASD
Return the object to user
With VLF
User requests object
A
Is user a "VLF user"
N
Y
B
Issue COFRETRI to retrieve object
C
Find and read the member
Object found ?
Y
N
Return object to user
UTOKEN valid ?
N
Y
Find and read the member
If necessary, allocate and OPEN DDNAME
D
Issue COFCREAT to create the object
C
Issue COFIDENT to obtain UTOKEN
Failure
Result ?
Success
B
class not defined
Issue COFDEFIN to define the class
A
set "not a VLF user"
N
Class defined ?
Y
D

Figure 47. Retrieving an Object for an End User

Defining a class of VLF objects

To define a class of VLF objects, issue the COFDEFIN macro. When you issue COFDEFIN, VLF uses information you specify on the macro and information from the active COFVLFxx member of SYS1.PARMLIB to prepare to process objects in that class. You must issue COFDEFIN for each class of objects that you want VLF to process.

To issue COFDEFIN specify:

- The class name. This name must match a class described on a CLASS statement in COFVLFxx.

- The length of the major names in the class. The default length (and only valid length) for a PDS class is 50.
- The length of the minor names in the class. The default length (and only valid length) for a PDS class is 8.

Note that you do not specify eligible major names on COFDEFIN. VLF obtains the eligible major names for the class from the CLASS statement in COFVLFxx.

If you want to make sure that only authorized code can retrieve objects, specify AUTHRET=YES when you define the class.

The CLASS statement in COFVLFxx defines the amount of virtual storage available for a class. On the COFDEFIN macro, however, you indicate how you want VLF to manage virtual storage for the objects in the class. If you specify TRIM=ON, which is the default, VLF automatically removes the least recently used objects when it needs space. If you specify TRIM=OFF, VLF removes objects only when it is specifically notified. Allowing VLF to manage the storage (TRIM=ON) ensures that, if space is limited, the most recently used objects tend to remain in virtual storage.

Identifying an end user to VLF

To identify an end user to VLF and to connect that end user to a VLF class, issue the COFIDENT macro. Your application must, to maintain the security of your installation's data, ensure that the user is authorized to access the objects. Before you issue COFIDENT, follow your system's security procedures (perhaps using RACF) to verify the user's authority to access the objects. For a PDS class, open the DDNAME you expect to specify on COFIDENT; otherwise, the COFIDENT will fail.

After verifying that the user is authorized to access the objects, issue COFIDENT. VLF connects the end user to a class and returns a user token that uniquely identifies the user of the class. VLF then ensures that only an identified user can access objects.

To issue COFIDENT, you must specify:

- The class name. For the identification to succeed, the class you name must already have been defined to VLF (through the COFDEFIN macro).
- The major name search order. On the DDNAME parameter (for a PDS class) or on the MAJNLIST parameter (for a non-PDS class), you specify the list of major names. The order in which these names are listed determines the search order for that particular user, and the search order can be different for each user. A single user can have more than one search order, but you must issue COFIDENT separately for each search order.

  This information comes from the user. For a PDS class, the user supplies the DDNAME of the partitioned data set concatenation. VLF determines the data set names in the list. (VLF allows private data sets in the list, but it creates objects only from major names that correspond to the major names specified for the class in the COFVLFxx parmlib member.)
- An area where VLF can return a user token (UTOKEN). The user token identifies this user as an identified user with access to a particular class of objects. Other VLF functions, such as retrieving an object or creating an object, require your application to supply the UTOKEN.

Your application design must allow you to keep track of the relationship between an end user, a class, and the token that connects the two. If a single user needs access to more than one class of VLF objects, you must issue COFIDENT for each class. An end user needs a unique user token for each class.

The SCOPE parameter on COFIDENT determines which tasks can retrieve objects with the returned user token. If you specify SCOPE=HOME, only a task running under the same home address space as the task that issued COFIDENT can retrieve objects with the returned UTOKEN. SCOPE=HOME is the default. If you specify SCOPE=SYSTEM, tasks running in any address space can retrieve objects with the returned UTOKEN.

The value you specify for SCOPE affects only the retrieving of objects with the COFRETRI macro. All other VLF macros that supply the user token must have the same home ASID as the user of COFIDENT.

Retrieving a VLF object

To retrieve a VLF object, issue the COFRETRI macro. Retrieving a VLF object means using VLF to obtain a copy of a VLF object on behalf of an end user. (Your application must check authority to access objects before you issue COFIDENT, the macro that connects the user to a class of VLF objects.) VLF retrieves objects according to the major name search order of the VLF end user, identified by the user token (UTOKEN).

When an end user requests a VLF object, your application must issue COFRETRI to attempt to retrieve the object before attempting to create the object. VLF ensures that, if you follow this processing order, VLF does not create an object if the permanent source data on DASD changes between the time you obtain the object from permanent storage and the time you create the object.

To issue COFRETRI, you must specify:

- The minor name of the object. This information comes from the user request.
- The user token (UTOKEN). This user token is the token returned from the earlier COFIDENT macro for the user. From the UTOKEN, VLF knows the class and major name search order for the user.

When you issue COFRETRI, you must provide locations where VLF can return the object it retrieves from virtual storage. You provide this area by defining a target area list, using two parameters: TLSIZE and TLIST.

TLSIZE defines the total size of the target area list. TLIST defines the target area list, which describes the target areas into which VLF is to place consecutive areas of the object. There must be at least one target area; there can be as many as sixteen target areas.

You must also provide locations where VLF can return:

- The size of the object it retrieves (OBJSIZE).
- The concatenation index (CINDEX) of the major name associated with the object. The index is the zero-origin relative number of the major name for the object in the major name list of the user.

  For concatenated partitioned data sets, the CINDEX value is the same as the “K” (concatenation index) value returned when a BLDL is performed to locate a member. This value indicates which major name in the concatenation list includes the object that VLF has retrieved.

When control returns to your program from COFRETRI, check the return codes carefully. Possible actions for some common return codes include:

- For return code 2 (best available object found), your application can issue a BLDL to determine if the object VLF returned is the correct object. If it is not, use traditional methods to obtain the object from DASD for the user, then issue the COFCREAT macro to create the object in VLF storage.
- For return code 4 (the object was found but the target area was too small), provide a larger target area and reissue COFRETRI.
- For return code 8 (no object was found), issue a BLDL and use traditional methods to obtain the object from DASD for the user, then issue the COFCREAT macro to create the object in VLF storage.

Creating a VLF object

To create a VLF object, obtain the object from DASD, then issue the COFCREAT macro. Creating a VLF object means using VLF to add an object to a class of VLF objects in virtual storage. You create a VLF object when an end user requests an object that does not yet exist in VLF storage.

To ensure that VLF does not create an object if the permanent source data on DASD changes between the time you obtain the object from permanent storage and the time you create the object, VLF requires that you issue COFRETRI to try to retrieve the object before you issue COFCREAT.

Thus, normal processing of an end user request for an object includes the following steps:

1. Issue the COFRETRI macro to attempt to retrieve the object.
2. Examine the return code from COFRETRI.
3. If the return code is 2 or 8, create the object. Between issuing the COFRETRI and the COFCREAT for this object, do not issue any COFRETRI macro with the same UTOKEN but a different minor name.

To ensure the integrity of the data, the working storage that your application uses to create the VLF object must not be key 8 storage, and you must perform the following steps:

1. Change to (or remain in) supervisor state.
2. Issue a BLDL macro for the PDS member using the same DDNAME used to identify the user to VLF. VLF guarantees that no manipulations with allocation can allow the user to alter the data sets associated with a DDNAME used to identify a VLF user. In such a case, VLF invalidates that user's token (UTOKEN).
3. Save the “K” value from a successful BLDL to pass to VLF as the CINDEX value on COFCREAT.
4. Perform secure I/O to read the object from DASD. Performing secure I/O, which protects the data from malicious tasks, has the following requirements:
   1. The DCB used for I/O must not be in key 8 storage.
   2. The I/O buffers must not be in key 8 storage.
5. Issue the COFCREAT macro to create the VLF object.
6. If necessary, copy the object to key 8 storage to enable the user program to access it.

Failure to follow these rules compromises the integrity of data objects in VLF storage. Depending on the nature of the class of VLF objects, incorrect data could cause severe system integrity problems.

To issue COFCREAT, you must specify:

- The major name for the object:
  - – For a non-PDS class, you specify the major name on the MAJOR parameter.
  - – For a PDS class, you specify the major name indirectly through the concatenation index on the CINDEX parameter. (You obtain this value by issuing a BLDL macro.)

  This information comes from the end user requesting the object.

  To ensure that VLF does not retain objects containing down-level data, your application must determine the highest level major name (in the user's major name search order) only after it receives a non-zero return code from COFRETRI. For a PDS class, you must issue the BLDL after the COFRETRI.
- The minor name of the object. This information comes from the end user requesting the object.
- The user token for the user on whose behalf you are creating this object. This token is the UTOKEN returned when you issued the COFIDENT macro for the user.

You must also describe where VLF can find the source data for the object it is to add to its virtual storage. You provide this source area for the object by defining an object parts list, using two parameters: OBJPLSZ and OBJPRTL.

OBJPLSZ defines the total size of the object parts list. OBJPRTL defines the object parts list, which describes source areas from which VLF can obtain consecutive pieces of the object. There must be at least one source area; there can be as many as sixteen source areas.

To issue COFCREAT, your program must be running under a task with the same home ASID as the issuer of the COFIDENT macro that identified the user.

Using REPLACE

For non-PDS classes, you can issue COFCREAT with the REPLACE option. If you specify REPLACE, VLF does not require that COFRETRI precede COFCREAT. Because VLF cannot then guarantee that the source object has not changed, your application must ensure that the source object remains unchanged between the time when you reference the source object to create the object parts list and the time when you receive control back from COFCREAT.

If you issue COFCREAT without REPLACE for an object that already exists in VLF storage, VLF returns a successful completion code but does not replace the data object. In this case, VLF assumes that the data object you supply is identical to the data object that already exists in VLF storage.

If you issue COFCREAT with REPLACE for an object that already exists in VLF storage, VLF does replace the existing object with the parts specified in the object parts list. In this case, VLF assumes that the data object you supply is more current than the data object that already exists in VLF storage.

Notifying VLF of a change

The system automatically informs VLF of all PDS updates made through IBM-supplied code. Thus, for PDS classes, your application does not need to notify VLF about PDS updates unless your installation uses its own private method of updating partitioned data sets.

If you need to notify VLF that an object, or some set of objects, in VLF storage is no longer valid, issue the COFNOTIF macro. You normally use COFNOTIF to inform VLF of changes to the permanent data kept on auxiliary storage devices, to ensure that the VLF objects reflect the current level of the permanent data on DASD.

To issue COFNOTIF, you must specify:

- The major name or a list of major names. The structure of the major name for PDS classes consists of the following elements:
  - 6-character volume serial number (padded with blanks if necessary)
  - PDS name (44-characters maximum), padded with blanks to equal the MAJLEN value.

  For example, where the volume serial number is VOL123 and the data set name is MYPDS, specify VOL123MYPDS, padded with blanks as required.
- The type of change you are reporting. A change to a minor name affects only the minor name within the major name specified on the MAJOR parameter.

  VLF views a minor name with one or more alias names as separate objects. Thus, to change a minor name, you must issue COFNOTIF for the minor name and for each alias name.

Specify the type of change with the FUNC parameter, as follows:

- FUNC=DELMAJOR — specifies that one or more major names have been deleted. VLF deletes all objects with those major names.
- FUNC=DELMINOR — specifies that one or more minor names have been deleted. VLF deletes any object with a matching minor name. With FUNC=DELMINOR, you must specify MINLIST to identify the minor name(s). When MAJOR is specified and the class is a non-PDS class, only matching minor names within that major name are deleted. Otherwise, all objects matching the minor name(s) in all PDS classes are deleted.
- FUNC=ADDMINOR — specifies that one or more minor names have been added to a major name. With FUNC=ADDMINOR, you must specify MINLIST to identify the minor name(s).
- FUNC=UPDMINOR — specifies that one or more objects corresponding to existing minor names have been changed. With FUNC=UPDMINOR, you must specify MINLIST to identify the minor name(s).
- FUNC=PURGEVOL — specifies that a physical storage device has been logically disconnected from the system, or that all of the information on the device has been deleted or replaced. With FUNC=PURGEVOL, you must specify VOLUME to identify the volume serial number of the volume.

For non-PDS classes, you must specify CLASS to restrict the changes to a single class. If you omit CLASS or use CLASS to specify the name of a PDS class, VLF assumes that the change affects all PDS classes.

Removing a VLF end user

To terminate an end user's access to a specific class of VLF objects, issue the COFREMOV macro. Issue COFREMOV when your application determines that an end user should no longer have access to the class of VLF objects. COFREMOV terminates an end user's access to the class of VLF objects associated with the specified user token (UTOKEN).

To issue COFREMOV, you must specify the user token (UTOKEN) returned from the COFIDENT macro that identified the user. To remove a user connected to more than one class, you must issue multiple COFREMOV macros.

You must issue COFREMOV from a task that has the same home ASID as the task that issued the COFIDENT to identify the user.

After you have removed the user, VLF rejects, with a reason code that indicates an unknown UTOKEN, any subsequent VLF requests that specify the UTOKEN.

Purging a VLF class

To request that VLF purge an entire class of objects previously defined to VLF, issue the COFPURGE macro. VLF deletes the class immediately. Any transaction in process for the purged class fails; VLF issues a failure return code that is appropriate for the transaction. Any user token (UTOKEN) connecting a user to the purged class becomes an invalid user token.

To issue COFPURGE, you must specify the class name.

If you then need to reinstate the class, you must issue another COFDEFIN for the class, which you can do at any time. Once you have reinstated the class, you must reidentify the users of the class.

Note that the system can also delete a class for control purposes even if no user requests it. You learn that a class has been purged when you issue a COFIDENT, COFREMOV, COFCREAT, or COFRETRI macro that specifies the purged class. There is a specific return and reason code combination for a class that is not defined or a UTOKEN that is not valid.

Modifying VLF

VLF can be modified using the MODIFY VLF,REPLACE,NN=*xx* command, instead of being stopped and restarted, so that its existing cache of objects can be retained. Use the command to start VLF with up to 16 concatenated COFVLF*xx* members of the logical parmlib, instead of the current members. The COFVLF*xx* parmlib members must contain valid CLASS statements.

The MODIFY VLF,REPLACE,NN=*xx* command can be used for the following actions:

- Deleting existing classes
- Adding new classes and their associated parameters
- Deleting EDSN or EMAJ major names from existing classes
- Adding EDSN or EMAJ major names to existing classes
- Increasing MaxVirt for existing classes
- Decreasing MaxVirt for existing classes
- Changing AlertAge for existing classes.

For further information about using the COFVLF*xx* member, refer to *z/OS MVS Initialization and Tuning Reference*.

For details on using the MODIFY VLF,REPLACE,NN=*xx* command, refer to *z/OS MVS System Commands*.

Chapter 13. Data-in-virtual

Data-in-virtual simplifies the writing of applications that use large amounts of data from permanent storage. Applications can create, read, and update data without the I/O buffer, blocksize, and record considerations that the traditional GET and PUT types of access methods require.

By using the services of data-in-virtual, certain applications that access large amounts of data can potentially improve their performance and their use of system resources. Such applications have an accessing pattern that is non-sequential and unpredictable. This kind of pattern is a function of conditions and values that are revealed only in the course of the processing. In these applications, the sequential record subdivisions of conventional access methods are meaningless to the central processing algorithm. It is difficult to adapt this class of applications to conventional record management programming techniques, which require all permanent storage access to be fundamentally record-oriented. Through the DIV macro, data-in-virtual provides a way for these applications to manipulate the data without the constraints of record-oriented processing.

An application written for data-in-virtual views its permanent storage data as a seamless body of data without internal record boundaries. By using the data-in-virtual MAP service, the application can make any portion of the object appear in virtual storage in an area called a **virtual storage window**. The window can exist in an address space, a data space, or a standard hiperspace. The application can reference and update the data in the window by using conventional processor instructions. To copy the updates to the object, the application uses the data-in-virtual SAVE service. For information about using data spaces or hiperspaces, see *z/OS MVS Programming: Extended Addressability Guide*.

An application written for data-in-virtual might also benefit by using the IARVSERV macro to share virtual storage, when that storage is in an address space or data space. For information about sharing data in virtual storage through IARVSERV, particularly the restrictions for using the data-in-virtual MAP and UNMAP services, see Chapter 16, "Sharing data in virtual storage (IARVSERV macro)," on page 309.

The data-in-virtual services process the application data in 4096-byte (4K-byte) units on 4K-byte boundaries called blocks. The application data resides in what is called a **data-in-virtual object,** a **data object,** or simply an **object**. The data-in-virtual object is a continuous string of uninterrupted data. The data object can be either a VSAM linear data set or a non-shared standard hiperspace. Choosing a linear data set as an object or a non-shared standard hiperspace as an object depends on your application. If your application requires the object to retain data, choose a linear data set, which provides permanent storage on DASD. A hiperspace object provides temporary storage.

When to use data-in-virtual

When an application reads more input and writes more output data than necessary, the unnecessary reads and writes impact performance. You can expect improved performance from data-in-virtual because it reduces the amount of unnecessary I/O.

As an example of unnecessary I/O, consider the I/O performed by an interactive application that requires immediate access to several large data sets. The program knows that some of the data, although not all of it, will be accessed. However, the program does not know ahead of time which data will be accessed. A possible strategy for gaining immediate access to all the data is to read all the data ahead of time, reading each data set in its entirety insofar as this is possible. Once read into main storage, the data can be accessed quickly. However, if only a small percentage of the data is likely to be accessed during any given period, the I/O performed on the unaccessed data is unnecessary.

Furthermore, if the application changes some data in main storage, it might not keep track of the changes. Therefore, to guarantee that all the changes are captured, the application must then write entire data sets back onto permanent storage even though only relatively few bytes are changed in the data sets.

Whenever such an application starts up, terminates, or accesses different data sets in an alternating manner, time is spent reading data that is not likely to be accessed. This time is essentially wasted, and the amount of it is proportional to the amount of unchanged data for which I/O is performed. Such applications are suitable candidates for a data-in-virtual implementation.

Factors affecting performance

When you write applications using the techniques of data-in-virtual, the I/O takes place only for the data referenced and saved. If you run an application using conventional access methods, and then run it again using data-in-virtual, you will notice a difference in performance. This difference depends on both the **size** of the data set and its **access pattern** (or reference pattern). Size refers to the magnitude of the data sets that the application must process. The access pattern refers to how the application references the data.

Engineering and scientific applications often use data access patterns that are suitable for data-in-virtual. Among the applications that can be considered for a data-in-virtual implementation are:

- Applications that process large arrays
- VSAM relative record applications
- BDAM fixed length record applications

Commercial applications sometimes use data access patterns that are not suitable because they are predictable and sequential. If the access pattern of a proposed application is fundamentally sequential or if the data set is small, a conventional VSAM (or other sequential access method) implementation may perform better than a data-in-virtual implementation. However, this does not rule out commercial applications as data-in-virtual candidates. If the performance factors are favorable, any type of application, commercial or scientific, is suitable for a data-in-virtual implementation.

Before you can use the DIV macro to process a linear data set object or a hiperspace object, you must create either the data set or the hiperspace. *z/OS MVS Programming: Extended Addressability Guide* explains how to create a hiperspace. “Creating a linear data set” explains how to create a linear data set.

Creating a linear data set

To create the data set, you need to specify the DEFINE CLUSTER function of IDCAMS with the LINEAR parameter. When you code the SHAREOPTIONS parameter for DEFINE CLUSTER, the cross-system value must be 3; that is, you may code SHAREOPTIONS as (1,3), (2,3), (3,3), or (4,3). Normally, you should use SHAREOPTIONS (1,3).

When creating a linear data set for data-in-virtual, you can use the LOCVIEW parameter of the DIV macro in conjunction with the other SHAREOPTIONS. LOCVIEW is described under the topic “The ACCESS service” on page 273. For a complete explanation of SHAREOPTIONS, see *z/OS DFSMS Using Data Sets*.

The following is a sample job that invokes Access Method Services (IDCAMS) to create the linear data set named DIV.SAMPLE on the volume called DIVPAK. When IDCAMS creates the data set, it creates it as an empty data set. Note that there is no RECORDS parameter; linear data sets do not have records.

```
//JNAME    JOB 'ALLOCATE LINEAR',MSGLEVEL=(1,1),
//         CLASS=R,MSGCLASS=D,USER=JOHNDOE
//*
//*        ALLOCATE A VSAM LINEAR DATASET
//*
//CLUSTPG  EXEC PGM=IDCAMS,REGION=4096K
//SYSPRINT DD SYSOUT=*
//DIVPAK   DD UNIT=3380,VOL=SER=DIVPAK,DISP=OLD
//SYSIN    DD *
           DEFINE CLUSTER (NAME(DIV.SAMPLE) -
                  VOLUMES(DIVPAK) -
                  TRACKS(1,1) -
                  SHAREOPTIONS(1,3) -
                  LINEAR)
/*
```

For further information on creating linear VSAM data sets and altering entry-sequenced VSAM data sets, see *z/OS DFSMS Access Method Services Commands*.

Using the services of data-in-virtual

Each invocation of the DIV macro requests any one of the services provided by data-in-virtual:

- IDENTIFY
- ACCESS
- MAP
- SAVE
- SAVELIST
- RESET
- UNMAP
- UNACCESS
- UNIDENTIFY

Identify

An application must use IDENTIFY to tell the system which data-in-virtual object it wants to process. IDENTIFY generates a unique ID, or token, that uniquely represents an application's request to use the given data object. The system returns this ID to the application. When the application requests other kinds of services with the DIV macro, the system supplies this ID as an input parameter. Specify DDNAME for a linear data set object and STOKEN for a hiperspace object.

An authorized application can also use IDENTIFY to request that data-in-virtual services bypass validity checking, and to assign ownership of the object's ID to another task.

Access

To gain the right to view or update the object, an application must use the ACCESS service. You normally invoke ACCESS after you invoke IDENTIFY and before you invoke MAP. ACCESS is similar to the OPEN macro of VSAM. It has a mode parameter of READ or UPDATE, and it gives your application the right to read or update the object.

If the object is a data set and if the SHAREOPTIONS parameter used to allocate the linear data set implies serialization, the system automatically serializes your access to the object. If access is not automatically serialized, you can serialize access to the object by using the ENQ, DEQ, and the RESERVE macros. If you do not serialize access to the object, you should consider using the LOCVIEW parameter to protect your window data against the unexpected changes that can occur when access to the object is not serialized. LOCVIEW is described under the topic "The ACCESS service" on page 273.

If the object is a hiperspace, DIV ensures that only one program can write to the object and that multiple users can only read the object. Only the task that owns the corresponding ID can issue ACCESS.

Map

The data object is stored in units of 4096-byte blocks. An application uses the MAP service to specify the part of the object that is to be processed in virtual storage. It can specify the entire object (all of the blocks), or a part of the object (any continuous range of blocks). Because parts of the same object can be viewed simultaneously through several different windows, the application can set up separate windows on the same object. However, a specific page of virtual storage cannot be in more than one window at a time.

After issuing ACCESS, the application obtains a virtual storage area large enough to contain the window. The size of the object, which ACCESS optionally returns, can determine how much virtual storage you need to request. After requesting virtual storage, the application invokes MAP. MAP establishes a one-to-one correspondence between blocks in the object and pages in virtual storage. A continuous range of pages corresponds to a continuous range of blocks. This correspondence is called a *virtual storage window*, or a *window*.

After issuing MAP, the application can look into the virtual storage area that the window contains. When it looks into this virtual storage area, it sees the same data that is in the object. When the application references this virtual storage area, it is referencing the data from the object. To reference the area in the window, the application simply uses any conventional processor instructions that access storage.

Although the object data becomes available in the window when the application invokes MAP, no actual movement of data from the object into the window occurs at that time. Actual movement of data from the object to the window occurs only when the application refers to data in the window. When the application references a page in the window for the first time, a page fault occurs. When the page fault occurs, the system reads the permanent storage block into real storage.

When the system brings data into real storage, the data movement involves only the precise block that the application references. The system updates the contents of the corresponding page in the window with the contents of the linear data set object. Thus, the system reads only the blocks that an application references into real storage. The sole exception to the system reading only the referenced blocks occurs when the application specifies LOCVIEW=MAP with the ACCESS service for a data set object.

Note:

1. If the application specifies LOCVIEW=MAP with ACCESS, the entire window is immediately filled with object data when the application invokes MAP.
2. If, when an application invokes MAP, it would rather keep in the window the data that existed before the window was established (instead of having the object data appear in the window), it can specify RETAIN=YES. Specifying RETAIN=YES is useful when creating an object or overlaying the contents of an object.
3. An important concept for data-in-virtual is the concept of *freshly obtained* storage. When virtual storage has been obtained and not subsequently modified, the storage is considered to be **freshly obtained**. The storage is also in this state when it has been obtained as a data space by using a DSPSERV CREATE and not subsequently modified. After a DSPSERV RELEASE, the storage is still considered freshly obtained until it has been modified. When referring to this storage or any of its included pages, this information uses *freshly obtained storage* and *freshly obtained pages*. If a program stores into a freshly obtained page, only that page loses its freshly obtained status, while other pages still retain it.
4. You can map virtual storage pages that are protected only when RETAIN=YES is specified. When the system establishes the virtual window, you can use the PGSER PROTECT macro to protect the data in the window. However, you must ensure that the data in the window is *not* protected when you issue the RESET form of the DIV macro.

Save, Savelist, and Reset

After using the MAP service, the application can access the data inside the window directly through normal programming techniques. When the application changes some data in the window, however, the data on the object does not consequently change. If the application wants the data changes in the window to appear in the object, it must use the SAVE service. SAVE writes, to the object, all changed blocks within the range to be saved inside the window. SAVE does not write unchanged blocks. When SAVE completes, the object contains any changes that the application made inside the virtual storage window. If a SAVE is preceded by another SAVE, the second SAVE will pick up only the changes that occurred since the previous SAVE.

Optionally, SAVE accepts a user list as input. To provide a user list, the application uses the SAVELIST service. SAVELIST returns the addresses of the first and last changed pages in each range of changed pages within the window. The application can then use these addresses as the user list for SAVE. The SAVE operation can be more efficient when using the list of addresses, so an application can improve its performance by using SAVELIST and then SAVE.

When specifying a user list and when a data space or hiperspace contains the window, the caller must use an STOKEN with SAVE to identify the data space or hiperspace.

If the application changes some data in a virtual storage window but then decides not to keep those changes, it can use the RESET service to reload the window with data from the object. RESET reloads only the blocks that have been changed unless you specify or have specified RELEASE=YES.

Unmap

When the application is finished processing the part of the object that is in the window, it eliminates the window by using UNMAP. To process a different part of the object, one not already mapped, the application can use the MAP service again. The SAVE, RESET, MAP, and UNMAP services can be invoked repeatedly as required by the processing requirements of the application.

If you issued multiple MAPs to different STOKENs, use STOKEN with UNMAP to identify the data space or hiperspace you want to unmap.

Note: If you do not want to retain the data in the virtual window, before you use the UNMAP service, use the PGSER UNPROTECT macro to “unprotect” any protected pages in the window.

If you issue UNMAP with RETAIN=NO and there are protected pages in the virtual storage window, the system loses the data in the protected pages and preserves the protection status. If you try to reference the protected pages, the system issues abend X'028'. To access the protected pages again, remove the protection status. Then issue the PGSER RELEASE or DSPSERV RELEASE macro to release all physical paging resources.

Unaccess

If the application has temporarily finished processing the object but still has other processing to perform, it uses UNACCESS to relinquish access to the object. Then other programs can access the object. If the application needs to access the same object again, it can regain access to the object by using the ACCESS service again without having to use the IDENTIFY service again.

Unidentify

UNIDENTIFY ends the use of a data-in-virtual object under a previously assigned ID that the IDENTIFY service returned.

The IDENTIFY service

To select the data-in-virtual object that you want to process, use IDENTIFY. IDENTIFY has the following required parameters:

- ID, TYPE, and DDNAME if TYPE=DA, or
- STOKEN if TYPE=HS.

IDENTIFY has two optional parameters: CHECKING and TTOKEN. Only programs running in supervisor state or under a PSW key of 0-7, or both, can use the CHECKING and TTOKEN parameters.

The following examples show two ways to code the IDENTIFY service.

Hiperspace™ object:

```
DIV IDENTIFY,ID=DIVOBJID,TYPE=HS,STOKEN=HSSTOK
```

Data set object:

```
DIV IDENTIFY,ID=DIVOBJID,TYPE=DA,DDNAME=DDAREA
```

ID: The ID parameter specifies the address where the IDENTIFY service returns a unique eight-byte name that connects a particular user with a particular object. This name is an output value from IDENTIFY, and it is also a required input value to all other services.

Simultaneous requests for different processing activities against the same data-in-virtual object can originate from different tasks or from different routines within the same task or the same routine. Each task or routine requesting processing activity against the object must first invoke the identify service. To correlate the various DIV macro invocations and processing activities, the eight-byte IDs generated by IDENTIFY are sufficiently unique to reflect the individuality of the IDENTIFY request, yet they all reflect the same data-in-virtual object.

TYPE: The TYPE parameter indicates the type of data-in-virtual object, either a linear data set (TYPE=DA) or a hiperspace (TYPE=HS). DIV does not support VSAM extended format linear data sets for use as a DIV object for which the size is greater than 4GB.

DDNAME: When you specify TYPE=DA for a data set object, you must specify DDNAME to identify your data-in-virtual object. If you specify TYPE=HS with IDENTIFY, do not specify DDNAME. (Specify STOKEN instead.) Do not specify a DDNAME that corresponds to a VSAM extended format linear data set for which the size is greater than 4GB, because DIV does not support them for use as a DIV object.

STOKEN: When you specify TYPE=HS for a hiperspace object, you must specify STOKEN to identify the hiperspace that is your data-in-virtual object. The STOKEN must be addressable in your primary address space. The hiperspace must be a standard hiperspace and must be owned by the task issuing the IDENTIFY. The system does not verify the STOKEN until your application uses the associated ID to access the object.

CHECKING: The CHECKING parameter enables you to request or bypass data-in-virtual validity checking. If you bypass validity checking, (CHECKING=NO), all data-in-virtual services bypass validity checking for the corresponding ID. You then have to ensure that the parameters and the environment are valid each time a data-in-virtual service is invoked. A parameter or environment that is not valid causes unpredictable results. The default is to allow data-in-virtual to perform validity checking. Bypass validity checking only if you need to improve data-in-virtual performance.

TTOKEN: To assign ownership of the corresponding ID to another TCB, code the TTOKEN parameter. You can assign the ID only to a TCB that is in the same TCB chain as your program, but is higher in the chain.

The ACCESS service

Your program uses the ACCESS service to request permission to read or update the object. ACCESS has two required parameters: ID and MODE, and two optional parameters: SIZE and LOCVIEW.

The following example shows one way to code the ACCESS service.

```
DIV ACCESS,ID=DIVOBJID,MODE=UPDATE,SIZE=OBJSIZE
```

ID: When you issue a DIV macro that requests the ACCESS service, you must also specify, on the ID parameter, the identifier that the IDENTIFY service returned. The task that issues ACCESS must own the corresponding ID. The ID parameter tells the system what object you want access to. When you request permission to access the object under a specified ID, the system checks the following conditions before it grants the access:

- You previously established the ID specified with your ACCESS request by invoking IDENTIFY.
- You have not already accessed the object under the same unique eight-byte ID. Before you can reaccess an already-accessed object under the same ID, you must first invoke UNACCESS for that ID.
- If your installation uses RACF, you must have the proper RACF authorization to access the object.
- If you are requesting read access, the object must not be empty. Use the MODE parameter to request read or update access.
- If the data object is a hiperspace, the system rejects the request if the hiperspace:
  - Has ever been the target of an ALESERV ADD.
  - Has one or more readers **and** one updater. (That is, the hiperspace can have readers and it can have one updater, but it can't have both.)
- If the data object is a linear data set, the system rejects the request if the linear data set:
  - Is a VSAM extended format linear data set for which the size is greater than 4GB.

MODE: The MODE parameter specifies how your program will access the object. You can specify a mode parameter of READ or UPDATE. They are described as follows:

- READ lets you read the object, but prevents you from using SAVE, to change the object.
- UPDATE, like READ, lets you read the object, but it also allows you update the object with SAVE.

Whether you specify READ or UPDATE, you can still make changes in the window, because the object does not change when you change the data in the window.

SIZE: The SIZE parameter specifies the address of the field where the system stores the size of the object. The system returns the size in this field whenever you specify SAVE or ACCESS with SIZE. If you omit SIZE or specify SIZE=*, the system does not return the size.

If you specified TYPE=DA with IDENTIFY for a data set object, SIZE specifies the address of a four-byte field. When control is returned to your program after the ACCESS service executes, the four-byte field contains the current size of the object. The size is the number of blocks that the application must map to ensure the mapping of the entire object.

If you specified TYPE=HS with IDENTIFY for a hiperspace object, ACCESS returns two sizes. The first is the current size of the hiperspace (in blocks). The second is the maximum size of the hiperspace (also in blocks). When specifying SIZE with an ID associated with a hiperspace object, you must provide an eight-byte field in which the system can return the sizes (4 bytes each).

LOCVIEW: The LOCVIEW parameter allows you to specify whether the system is to create a local copy of the data-in-virtual object.

If your object is a hiperspace, you cannot specify LOCVIEW=MAP.

If your object is a data set, you can code the LOCVIEW parameter two ways:

- LOCVIEW=MAP
- LOCVIEW=NONE (the default if you do not specify LOCVIEW)

If another program maps the same block of a data-in-virtual object as your program has mapped, a change in the object due to a SAVE by the other program can sometimes appear in the virtual storage window of your program. The change can appear when you allocate the data set object with a SHAREOPTIONS(2,3), SHAREOPTIONS(3,3), or SHAREOPTIONS(4,3) parameter, and when the other program is updating the object while your program is accessing it.

If the change appears when your program is processing the data in the window, processing results might be erroneous because the window data at the beginning of your processing is inconsistent with the window data at the end of your processing. For an explanation of SHAREOPTIONS, see *z/OS DFSMS Using Data Sets*. The relationship between SHAREOPTIONS and LOCVIEW is as follows:

- When you allocate the data set object by SHAREOPTIONS(2,3), SHAREOPTIONS(3,3), or SHAREOPTIONS(4,3), the system does not serialize the accesses that programs make to the object. Under these options, if the programs do not observe any serialization protocol, the data in your virtual storage window can change when other programs invoke SAVE. To ensure that your program has a consistent view of the object, and protect your window from changes that other programs make on the object, use LOCVIEW=MAP. If you do not use LOCVIEW=MAP when you invoke ACCESS, the system provides a return code of 4 and a reason code of hexadecimal 37 as a reminder that no serialization is in effect even though the access was successful.
- When you allocate the object by SHAREOPTIONS(1,3), object changes made by the other program cannot appear in your window because the system performs automatic serialization of access. Thus, when any program has update access to the object, the system automatically prevents all other access. Use LOCVIEW=NONE when you allocate the data set by SHAREOPTIONS(1,3).

  Note: The usual method of programming data-in-virtual is to use LOCVIEW=NONE and SHAREOPTIONS(1,3). LOCVIEW=MAP is provided for programs that must access a data object simultaneously. Those programs would not use SHAREOPTIONS(1,3).

LOCVIEW=MAP requires extra processing that degrades performance. Use LOCVIEW=NONE whenever possible although you can use LOCVIEW=MAP for small data objects without significant performance loss. When you write a program that uses LOCVIEW=MAP, be careful about making changes in the object size. Consider the following:

- When a group of programs, all using LOCVIEW=MAP, have simultaneous access to the same object, no program should invoke any SAVE or MAP that extends or truncates the object unless it informs the other programs by some coding protocol of a change in object size. When the other programs are informed, they can adjust their processing based on the new size.
- All the programs must create their maps before any program changes the object size. Subsequently, if any program wants to reset the map or create a new map,

it must not do so without observing the protocol of a size check. If the size changed, the program should invoke UNACCESS, followed by ACCESS to get the new size. Then the program can reset the map or create the new map.

The MAP service

The MAP service makes an association between part or all of an object, the portion being specified by the OFFSET and SPAN parameters, and your program's virtual storage. This association, which is called a **virtual storage window,** takes the form of a one-to-one correspondence between specified blocks on the object and specified pages in virtual storage. After the MAP is complete, your program can then process the data in the window. The RETAIN parameter specifies whether data that is in the window when you issue MAP is to remain or be replaced by the data from the associated object.

Note: You cannot map virtual storage pages that are page-fixed into a virtual storage window. Once the window exists, you can page-fix data inside the window so long as it is not fixed when you issue SAVE, UNMAP, or RESET.

If your window is in an address space, you can map either a linear data set or a hiperspace object. See Figure 48.

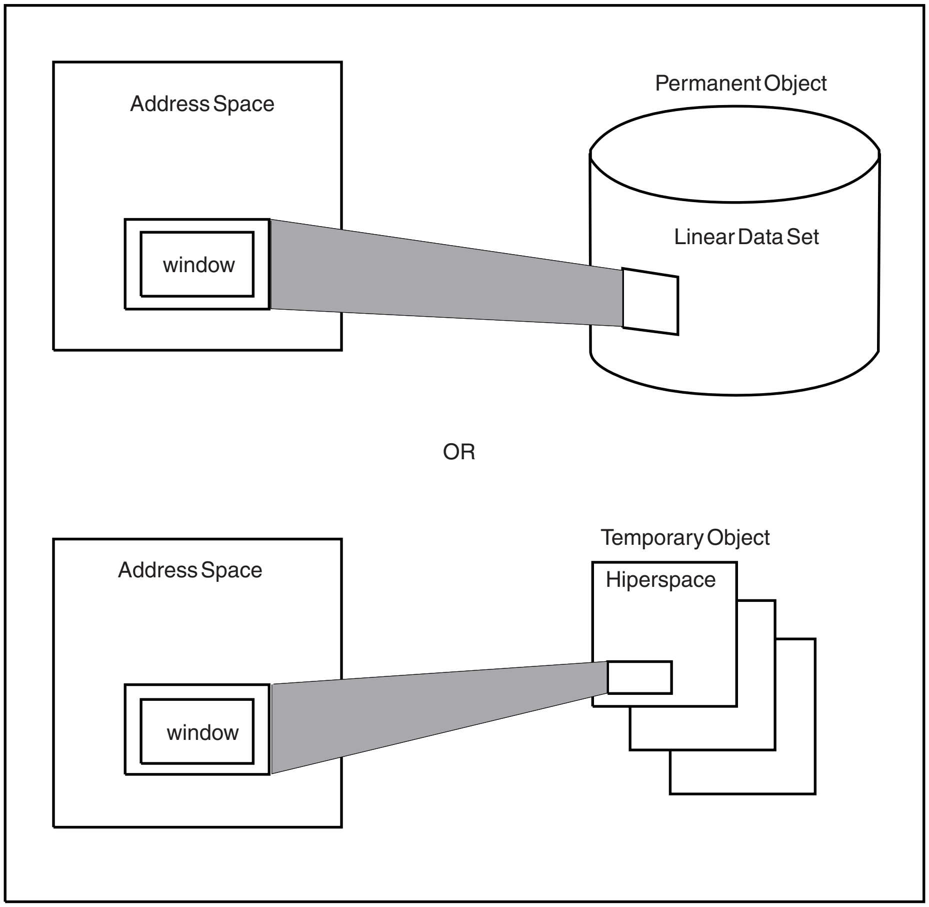


Figure 48. Mapping from an Address Space

If your window is in a data space or a hiperspace, you can map only to a linear data set. See Figure 49 on page 277.

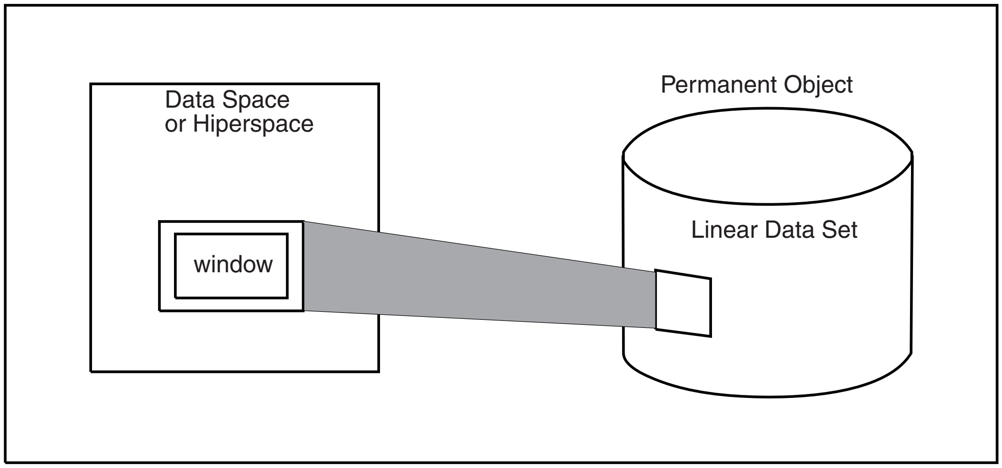


Figure 49. Mapping from a Data Space or Hiperspace

If your window is in a data space or hiperspace, you can issue multiple MAPs under the same ID from different data spaces or hiperspaces. You cannot mix data space maps or hiperspace maps with address space maps under the same ID at any one time. However, you can mix data space maps and hiperspace maps. See Figure 50 on page 278.

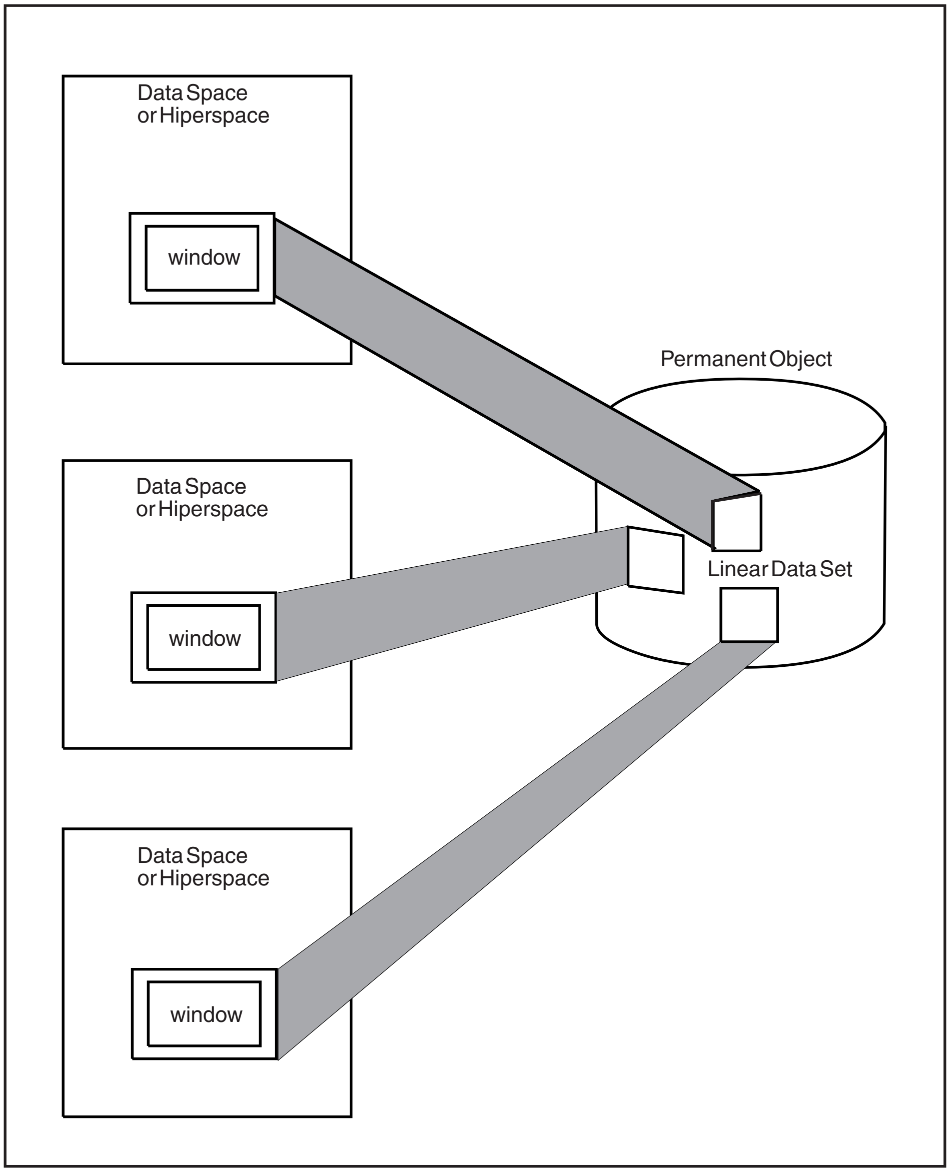


Figure 50. Multiple Mapping

The MAP service has two required parameters: ID and OFFSET, and five optional parameters: SPAN, AREA, RETAIN, STOKEN, and PFCOUNT.

The following examples show two ways to code the MAP service.

Hiperspace or data set object:

```
DIV MAP,ID=DIVOBJID,AREA=MAPPTR1,SPAN=SPANVAL,OFFSET=*,PFCOUNT=7
```

Data set object:

```
DIV MAP,ID=DIVOBJID,AREA=MAPPTR1,SPAN=SPANVAL,OFFSET=*,STOKEN=DSSTOK,PFCOUNT=7
```

ID: The ID parameter specifies the storage location containing the unique eight-byte value that was returned by IDENTIFY. The map service uses this value to determine which object is being mapped under which request.

If you specify the same ID on multiple invocations of the MAP service, you can create simultaneous windows corresponding to different parts of the object. However, an object block that is mapped into one window cannot be mapped into any other window created under the same ID. If you use different IDs, however, an object block can be included simultaneously in several windows.

OFFSET and SPAN: The OFFSET and SPAN parameters indicate a range of blocks on the object. Blocks in this range appear in the window. OFFSET indicates the first object block in the range, while SPAN indicates how many contiguous blocks make up the range. An offset of zero indicates the beginning of the object. For example, an offset of zero and a span of ten causes the block at the beginning of the object to appear in the window, together with the next nine object blocks. The window would then be ten pages long.

Specifying OFFSET=* or omitting OFFSET causes the system to use a default OFFSET of zero. Specifying SPAN=0, SPAN=*, or omitting SPAN results in a default SPAN of the number of blocks needed to MAP the entire object, starting from the block indicated by OFFSET. Specifying both OFFSET=* and SPAN=* or omitting both causes the entire object to appear in the window.

You may use the OFFSET and SPAN parameters to specify a range spanning any portion of the object, the entire object, or extending beyond the object. Specifying a range beyond the object enables a program to add data to the object, increasing the size of the object. If data in a mapped range beyond the object is saved (using the SAVE service), the size of the object is updated to reflect the new size.

To use the OFFSET parameter, specify the storage location containing the block offset of the first block to be mapped. The offset of the first block in the data object is zero. To use the SPAN parameter, specify the storage location containing the number of blocks in the mapped range.

AREA: When you specify MAP, you must also specify an AREA parameter. AREA indicates the beginning of a virtual storage space large enough to contain the entire window. Before invoking MAP, you must ensure that your task owns this virtual storage space. The storage must belong to a single, pageable private area subpool. It must begin on a 4096-byte boundary (that is, a page boundary) and have a length that is a multiple of 4096 bytes.

Note that any virtual storage space assigned to one window cannot be simultaneously assigned to another window. If your MAP request specifies a virtual storage location, via the AREA parameter, that is part of another window, the system rejects the request.

You cannot free virtual storage that is mapped into a window as long as the map exists. Attempts to do this will cause your program to abend. Subsequent attempts to reference the mapped virtual space will cause an ABEND.

RETAIN: The RETAIN parameter determines what data you can view in the window. It can be either the contents of the virtual storage area (that corresponds to the window) the way it was before you invoked MAP, or it can be the contents of the object. The following table shows how using the RETAIN parameter with MAP affects the contents of the window.

RETAIN=	**Window view**
NO (default)	Contents of mapped object

RETAIN=	**Window view**
YES	Contents of virtual storage

If you specify RETAIN=NO, or do not specify the RETAIN parameter at all (which defaults to RETAIN=NO), the contents of the object replace the contents of the virtual storage whenever your program references a page in the window. Virtual storage that corresponds to a range beyond the end of the object appears as binary zeroes when referenced. You can use RETAIN=NO to change some data and save it back to the object.

If you specify RETAIN=YES, the window retains the contents of virtual storage. The contents of the window are not replaced by data from the object. If you issue a subsequent SAVE, the data in the window *replaces* the data on the object. If the window extends beyond the object and your program has not referenced the pages in the extending part of the window, the system does not save the extending pages. However, if your program has referenced the extending pages, the system does save them on the object, extending the object so it can hold the additional data.

You can also use RETAIN=YES to reduce the size of (truncate) the object. If the part you want to truncate is mapped with RETAIN=YES and the window consists of freshly obtained storage, the data object size is reduced at SAVE time.

If you want to have zeroes written at the end of the object, the corresponding virtual storage must be explicitly set to zero prior to the SAVE.

STOKEN: To reference an entire linear data set through a single window, a program might require a considerable amount of virtual storage. In this case, the program can use a data space or hiperspace to contain the window. If you want the virtual storage window to be in a data space or hiperspace, specify STOKEN when you invoke MAP. When you specify STOKEN, you provide an eight-byte input parameter that identifies the data space or hiperspace, and that was returned from DSPSERV CREATE.

However, do not place the window in a data space or hiperspace under the following circumstances:

- If the data space is a disabled reference (DREF) data space.
- If the object is accessed with LOCVIEW=MAP.
- If the data space or hiperspace belongs to another task. However, if your program is in supervisor state or has a system storage key, it can use a data space or hiperspace that belongs to another task provided that the other task is in the same primary address space as your program.

PFCOUNT: PFCOUNT is useful for referencing sequential data. Because you get a page fault the first time you reference each page, preloading successive pages decreases the number of page faults.

The PFCOUNT parameter (*nnn*) is an unsigned decimal number up to 255. When an application references a mapped page, PFCOUNT tells the system that the program will be referencing this object in a sequential manner. PFCOUNT might improve performance because it asks the system to preload *nnn* pages, if possible. The system reads in *nnn* successive pages only to the end of the virtual range of the mapped area containing the originally referenced page, and only as resources are available.

You can use REFPAT INSTALL to define a reference pattern for the mapped area. In response to REFPAT, the system brings multiple pages into central storage when referenced. In this case, the PFCOUNT value you specify on DIV is not in effect as long as the reference pattern is in effect. When REFPAT REMOVE removes the definition of the reference pattern, the PFCOUNT you specify on DIV is again in effect. For information on the REFPAT macro, see *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

The SAVE service

The SAVE service writes changed pages from the window to the object if the changed pages are within the range to be saved. When you invoke SAVE, you specify one of the following:

- A single and continuous range of blocks in the data-in-virtual object with the use of OFFSET and SPAN. Any virtual storage windows inside this range are eligible to participate in the save.
- A user list supplied with the use of LISTADDR and LISTSIZE. The list must contain the addresses of the first and last changed pages for each range of changed pages within the window. The SAVELIST service can provide these addresses for the user list.

For a SAVE request to be valid, the object must currently be accessed with MODE=UPDATE, under the same ID as the one specified on this SAVE request. Because you can map an object beyond its current end, the object might be extended when the SAVE completes if there are changed pages beyond the current end at the time of the ACCESS. On the other hand, the SAVE truncates the object if freshly obtained pages are being saved that are mapped in a range that extends to or beyond the end of the object **and** additional non-freshly-obtained pages beyond the object area are not also being saved. In either case, the new object size is returned to your program if you specify the SIZE parameter.

When the system writes the pages from the window to the object, it clears (sets to zeroes) blocks in the object that are mapped to freshly obtained pages in the window if either one of the following conditions is true:

- There are subsequent pages in the range being saved that are not freshly obtained
- The blocks mapped to the freshly obtained pages are not at the end of the object. That is, they are imbedded in the object somewhere before the last block of the object. If the blocks mapped to freshly obtained pages do extend to the end of the object and no subsequent non-freshly obtained pages are being saved, then the object is truncated by that number of blocks.

If you specified RETAIN=YES with MAP, SAVE treats pages in the window that you have not previously saved as changed pages and will write them to the object.

Note:

1. Do not use SAVE for a storage range that contains DREF or page fixed storage.
2. If data to be saved has not changed since the last SAVE, no I/O will be performed. The performance advantages of using data-in-virtual are primarily because of the automatic elimination of unnecessary read and write I/O operations.
3. The range specified with SAVE can extend beyond the end of the object.
4. The system does not save pages of an object that was not mapped to any window.

5. The system does not save pages in a window that lies outside the specified range.

The following example shows how to code the SAVE service for a hiperspace or data set object.

```
DIV SAVE,ID=DIVOBJID,SPAN=SPAVAL,OFFSET=*,SIZE=OBJSIZE
```

ID: The ID parameter tells the SAVE service which data object the system is writing to under which request. Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY. You must have previously accessed the object with MODE=UPDATE under the same ID as the one specified for SAVE.

OFFSET and SPAN: Use the OFFSET and SPAN parameters to select a continuous range of object blocks within which the SAVE service can operate. OFFSET indicates the first block and SPAN indicates the number of blocks in the range. As in the MAP service, the offset and span parameters refer to object blocks; they do not refer to pages in the window. You cannot specify OFFSET and SPAN when you specify LISTADDR and LISTSIZE.

Specifying OFFSET=* or omitting OFFSET causes the system to use the default offset (zero). The zero offset does not omit or skip over any of the object blocks, and it causes the range to start right at the beginning of the object. Specifying SPAN=0, SPAN=*, or omitting SPAN gives you the default span. The default span includes the first object block after the part skipped by the offset, and it includes the entire succession of object blocks up to and including the object block that corresponds to the last page of the last window.

When SAVE executes, it examines each virtual storage window established for the object. In each window, it detects every page that corresponds to an object block in the selected range. Then, if the page has changed since the last SAVE, the system writes the page on the object. (If the page has not changed since the last SAVE, it is already identical to the corresponding object block and there is no need to save it.) Although SAVE discriminates between blocks on the basis of whether they have changed, it has the effect of saving all window pages that lie in the selected range. Specifying both OFFSET=* and SPAN=* or omitting both causes the system to save all changed pages in the window without exceptions.

To use the OFFSET parameter, specify the storage location containing the block offset of the first block to be saved. The offset of the first block in the object is zero. To use the SPAN parameter, specify the storage location containing the number of blocks in the range to be saved.

SIZE: When you specify SIZE after the SAVE completes, the system returns the size of the data object in the virtual storage location specified by the SIZE parameter. If you omit SIZE or specify SIZE=*, the system does not return the size value. If TYPE=DA, invoking SAVE can change the size of the object. If TYPE=HS, invoking SAVE has no effect on the size of the object.

LISTADDR: The LISTADDR parameter specifies the address of the first entry in the user list. Use this parameter and the LISTSIZE parameter when you specify a user list as input for SAVE.

LISTSIZE: The LISTSIZE parameter specifies the number of entries in the user list. Use this parameter and the LISTADDR parameter when you specify a user list as input for SAVE.

STOKEN: If you specify a user list as input for SAVE and a data space or hiperspace contains the window, you must specify STOKEN when you invoke SAVE. When you specify STOKEN, you provide an eight-byte input parameter that identifies the data space or hiperspace, and that was returned from DSPSERV CREATE.

The SAVELIST service

The advantage of using SAVELIST with SAVE is improved performance, especially for applications that manipulate graphic images. The SAVELIST service allows the application to inspect and verify data only in pages that have been changed. In a user list provided by the application, SAVELIST returns the addresses of the first and last page in each range of changed pages within the window. The mapped ranges may be either address spaces, data spaces or hiperspaces. If more than one data space or hiperspace is mapped onto a DIV object, the selected range must be contained within a single data space or hiperspace.

The application must set up a user list before issuing SAVELIST. Upon return from SAVELIST, the first word of each list entry holds the virtual storage address of the first page in a range of changed pages. The second word of the entry holds the virtual storage address of the last changed page in this range. In the last valid entry of the user list, the high-order bit of the first word is set to one.

If the reason code indicates that there are more changed pages that can fit in this list, the first word of the last entry in the list contains an offset (in block number format) into the DIV object from which more changed pages might exist. The second word of the last entry contains the span from the new offset to the block pointed to by the original OFFSET/SPAN combination. If more changed pages can fit in the user list, you can issue SAVE with the current list, and then issue SAVELIST and SAVE again to obtain the addresses of the additional changed pages and to save them.

ID: Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY, which connects a particular user to a particular object.

LISTADDR: The LISTADDR parameter specifies the address of the first entry in the user list.

LISTSIZE: The LISTSIZE parameter specifies the number of entries in the list. The size of the list must be a minimum of three entries and a maximum of 255 entries. The SAVELIST service can place addresses in all but the last two entries, which the macro uses as a work area.

The RESET service

At times during program processing, your program might have made changes to pages in the virtual storage window, and might no longer want to keep those changes. RESET, which is the opposite of SAVE, replaces data in the virtual storage window with data from the object. As with SAVE and MAP, the range to be reset refers to the object rather than the virtual storage. RESET resets only windows that are within the specified range, and it resets all the windows in the range that your program changed.

Do not use RESET for a storage range that contains DREF storage.

Effect of RETAIN mode on RESET

You actually specify RETAIN on MAP, not on RESET, but the RETAIN mode of each individual window affects how the system resets the window. The following table shows the effect that issuing RETAIN with MAP has on RESET.

RETAIN=	RESET results
NO (default)	The data in the window matches the object data as of the last SAVE.
YES	Unless saved, the data in the window become freshly obtained. Any pages previously saved re-appear in their corresponding window. All other pages appear freshly obtained.

The system resets the window as follows:

- If you specified RETAIN=NO with MAP, after the RESET, the data in the window matches the object data as of the last SAVE. This applies to all the pages in the window.
- If you specified RETAIN=YES with MAP, the pages in the window acquire a freshly obtained status after the RESET unless you have been doing SAVE operations on this window. Individual object blocks changed by those SAVE operations re-appear after the RESET in their corresponding window pages, together with the other pages. However, the other pages appear freshly obtained.

Note: Regardless of the RETAIN mode of the window, any window page that corresponds to a block beyond the end of the object appears as a freshly obtained page.

The following example shows how to code the RESET service for a hiperspace or data set object:

```
DIV RESET,ID=DIVOBJID,SPAN=SPANVAL,OFFSET=*,RELEASE=YES
```

ID: The ID parameter tells the RESET service what data object is being written to. Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY and used with previous MAP requests. You must have previously accessed the object (with either MODE=READ or MODE=UPDATE) under the same ID as the one currently specified for RESET.

OFFSET and SPAN: The OFFSET and SPAN parameters indicate the RESET range, the part of the object that is to supply the data for the RESET. As with MAP and SAVE, OFFSET indicates the first object block in the range, while SPAN indicates how many contiguous blocks make up the range, starting from the block indicated by OFFSET. The first block of the object has an offset of zero.

To use OFFSET, specify the storage location containing the block offset of the first block to be reset. To use SPAN, specify the storage location containing the number of blocks in the range to be RESET. Specifying OFFSET=* or omitting OFFSET causes the system to use a default OFFSET of zero. Specifying SPAN=* or omitting SPAN sets the default to the number of blocks needed to reset all the virtual storage windows that are mapped under the specified ID starting only with the block number indicated by OFFSET. Specifying both OFFSET=* and SPAN=* or omitting both resets all windows that are currently mapped under the specified ID.

RELEASE: RELEASE=YES tells the system to release all pages in the reset range. RELEASE=NO does not replace unchanged pages in the window with a new copy of pages from the object. It replaces only changed pages. Another ID might have changed the object itself while you viewed data in the window. Specify RELEASE=YES to reset all pages. Any subsequent reference to these pages causes the system to load a new copy of the data page from the object.

Your program uses the UNMAP service to remove the association between a window in virtual storage and the object. Each UNMAP request must correspond to a previous MAP request. Note that UNMAP has no effect on the object. If you made changes in virtual storage but have not yet saved them, the system does not save them on the object when you issue UNMAP. UNMAP has two required parameters: ID and AREA, and two optional parameters: RETAIN and STOKEN.

The following examples show two ways to code the UNMAP service.

Hiperspace or data set object:

```
DIV UNMAP,ID=DIVOBJID,AREA=MAPPTR1
```

Data set object:

```
DIV UNMAP,ID=DIVOBJID,AREA=MAPPTR1,STOKEN=DSSTOK
```

ID: The ID parameter you specify is the address of an eight-byte field in storage. That field contains the identifier associated with the object. The identifier is the same value that the IDENTIFY service returned, which is also the same value you specified when you issued the corresponding MAP request.

AREA: The AREA parameter specifies the address of a four-byte field in storage that contains a pointer to the start of the virtual storage to be unmapped. This address must point to the beginning of a window. It is the same address that you provided when you issued the corresponding MAP request.

RETAIN: RETAIN specifies the state that virtual storage is to be left in after it is unmapped, that is, after you remove the correspondence between virtual storage and the object.

Specifying RETAIN=NO with UNMAP indicates that the data in the unmapped window is to be freshly obtained.

If your object is a hiperspace, you cannot specify RETAIN=YES. If your object is a data set, you can specify RETAIN=YES.

Specifying RETAIN=YES on the corresponding UNMAP transfers the data of the object into the unchanged pages in the window. In this case, RETAIN=YES with UNMAP specifies that the virtual storage area corresponding to the unmapped window is to contain the last view of the object. After UNMAP, your program can still reference and change the data in this virtual storage but can no longer save it on the object unless the virtual area is mapped again.

Note:

1. If you issue UNMAP with RETAIN=NO, and there are unsaved changes in the virtual storage window, those changes are lost.
2. If you issue UNMAP with RETAIN=YES, and there are unsaved changes in the window, they remain in the virtual storage.
3. Using UNMAP with RETAIN=YES has certain performance implications. It causes the system to read unreferenced pages, and maybe some unchanged ones, from the object. You must not unmap with RETAIN=YES if your object is a hiperspace.

4. If the window is in a deleted data space, UNMAP works differently depending on whether you specify RETAIN=YES or RETAIN=NO. If you specify RETAIN=YES, the unmap fails and the program abends. Otherwise, the unmap is successful.

STOKEN: If you issued multiple maps under the same ID with different STOKENs, use STOKEN with UNMAP. If you do not specify STOKEN in this case, the system will scan the mapped ranges and unmap the first range that matches the specified virtual area regardless of the data space it is in. Issuing UNACCESS or UNIDENTIFY automatically unmaps all mapped ranges.

The UNACCESS and UNIDENTIFY services

Use UNACCESS to terminate your access to the object for the specified ID. UNACCESS automatically includes an implied UNMAP. If you issue an UNACCESS with outstanding virtual storage windows, any windows that exist for the specified ID are unmapped with RETAIN=NO. The ID parameter is the sole parameter of the UNACCESS service, and it designates the same ID that you specified in the corresponding ACCESS. As in the other services, use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY.

Use UNIDENTIFY to notify the system that your use of an object under the specified ID has ended. If the object is still accessed as an object under this ID, UNIDENTIFY automatically includes an implied UNACCESS. The UNACCESS, in turn, issues any necessary UNMAPs using RETAIN=NO. The ID parameter is the only parameter for UNIDENTIFY, and it must designate the same ID as the one specified in the corresponding ACCESS. As in the other services, use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY.

The following example shows how to code the UNACCESS and UNIDENTIFY services for a hiperspace or data set object:

```
DIV UNACCESS,ID=DIVOBJID
DIV UNIDENTIFY,ID=DIVOBJID
```

Sharing data in an object

The services of data-in-virtual are task-oriented. When a user issues IDENTIFY, an association is established between the ID assigned and the user's task. The type of association differs, however, depending on whether the task is authorized or not. The authorized task runs in supervisor state, has a system key (0-7), or has APF authorization. The unauthorized task runs in problem program state with a user key and with no APF authorization.

- For the unauthorized user, using data-in-virtual services for a specific ID is strictly local to its immediate task. That is, all services for a particular ID must be requested by the same task that requested IDENTIFY and obtained the ID.
- For the authorized user, one task can issue IDENTIFY, ACCESS, and UNACCESS while authorized subtasks of that task can request all the other services by using the ID returned by IDENTIFY. If you specified TYPE=HS with IDENTIFY, you cannot issue MAP or SAVE unless your task is the same as the one that issued IDENTIFY.

Any task, authorized or not, can reference or change the data in a mapped virtual storage window, even if the window was mapped by another task, and even if the object was identified and accessed by another task. As long as the task can address the window, it is allowed to reference or change the included data. However, only the task that issued the IDENTIFY can issue the SAVE to change the object.

When more than one user has the ability to change the data in one storage area, you must take steps to serialize their use of the area.

Because data-in-virtual services affect virtual storage, the PSW key of any task that requests a service (under a given ID) must be the same as the PSW key of the task that issued the IDENTIFY (that obtained the ID). This is not required if the task has PSW key zero.

DIV macro programming examples

The following programming examples illustrate how to code and execute a program that processes a data set data-in-virtual object.

Executing an application

The following JCL job executes an application program called SAMPLE. SAMPLE performs some kind of application-oriented processing on the dataset object, DIV.SAMPLE, that was allocated in “Creating a linear data set” on page 268.

When SAMPLE executes, it issues a DIV macro specifying to the IDENTIFY service the name of the data-in-virtual object that it will process. To identify the data set, SAMPLE specifies the ddname, DYNAMIC, on the DDNAME parameter of the DIV macro.

The system then connects the actual data set name, DIV.SAMPLE, with the program that will process it. The link between the application program and the data set is the name, DYNAMIC, which appears in both the application and the JCL.

```
//*
//*
//*        EXECUTE A DATA-IN-VIRTUAL APPLICATION
//*
//SAMPLE    EXEC  PGM=SAMPLE
//STEPLIB   DD DSN=DIV22.LOAD.JOBS,DISP=SHR
//DYNAMIC   DD DSN=DIV.SAMPLE,DISP=SHR
//SYSABEND  DD  SYSOUT=*
/*
```

Processing a data-in-virtual object

The following example shows a program that processes a data-in-virtual object. The first part of the program identifies the data set and accesses the object. Then it obtains the virtual storage where it will place the window.

```
SAMPLE   CSECT ,
SAMPLE   AMODE 31
SAMPLE   RMODE ANY
*
*        FUNCTION: OBTAIN VIRTUAL STORAGE.  THEN IDENTIFY AND
*        ACCESS THE LINEAR DATA SET.  THEN MAP AND PROCESS THE
*        VIRTUAL STORAGE, AND STORE DATA INTO IT. THEN DO SAVES &
*        RESETS.  FINISH UP WITH AN UNMAP, AN UNACCESS AND AN
*        UNIDENTIFY.  ALL INVOCATIONS OF DATA-IN-VIRTUAL IN THIS
*        PROGRAM DEFAULT TO 'RETAIN = NO'.
*
*        DESCRIPTION: THIS JOB MAKES CHANGES IN THE LINEAR DATASET
*        CLUSTER, 'DIV.SAMPLE', WHICH IS TREATED AS A LINEAR
*        DATASET.  AFTER THIS JOB IS RUN, THE DATASET WILL CONTAIN
*        SEVEN PAGES OF ONES, FOLLOWED BY ONE PAGE OF ZEROES,
*        FOLLOWED BY EIGHT PAGES OF FIVES.  IT IS ASSUMED THE
*        DATASET WAS CREATED BY A DEFINE CLUSTER COMMAND AND THAT
*        IT CONTAINS ZEROES WHEN THIS PROGRAM BEGINS TO EXECUTE.
*
@MAINENT DS    0H
         USING *,R15
         B     @PROLOG
         DC    AL1(14)
         DC    C'SAMPLE PROGRAM'
         DROP  R15
@PROLOG  STM   R14,R12,12(R13)    STD ENTRY LINKAGE
         LR    R12,R15
         USING SAMPLE,R12
         ST    R13,SAVEAREA+4
         LR    R2,R13
         LA    R13,SAVEAREA
         ST    R13,8(R2)
         SR    R15,R15            CLEAR R15
*
*        GET STORAGE FOR THE WINDOW
*
         GETMAIN RU,LV=16*4096,SP=0,BNDRY=PAGE
         ST    R1,MAPPTR1         PTR TO STORAGE
*
*        INVOKE IDENTIFY SERVICE OF DIV MACRO
*
         DIV   IDENTIFY,DDNAME=DDAREA,TYPE=DA,ID=TESTID
         LTR   R15,R15            CHECK IF RC IS ZERO
         BNZ   ERROR              IDENTIFY FAILED
*
*        INVOKE ACCESS SERVICE OF DIV MACRO
*
         DIV   ACCESS,MODE=UPDATE,ID=TESTID
         LTR   R15,R15            CHECK IF RC IS ZERO
         BNZ   ERROR              ACCESS FAILED
```

The program maps the data set object. The resulting virtual storage window is eight pages long, and it corresponds to the second eight blocks of the object. The window is situated in the virtual storage obtained earlier by the GETMAIN macro. The program fills the window with fives, then saves the window back into the second eight blocks of the object. The program eliminates the window by invoking UNMAP.

```
*        INVOKE THE MAP SERVICE OF THE DIV MACRO
*        TO SKIP THE FIRST EIGHT BLOCKS OF THE OBJECT
*
         L     R3,EIGHT           GET SPAN
         ST    R3,SPVALUE         INITIALIZE SPAN
         ST    R3,OFFS            INITIALIZE OFFSET
         DIV   MAP,ID=TESTID,AREA=MAPPTR1,               x
               SPAN=SPVALUE,OFFSET=OFFS
         LTR   R15,R15            CHECK IF RC IS ZERO
         BNZ   ERROR              MAP FAILED
*
*        FILL IN  5'S FOR ALL EIGHT MAPPED BLOCKS
*
         L     R1,MAPPTR1         POINTS TO WINDOW
         LR    R2,R1              POINTS TO MAP
         SR    R5,R5              COUNTER 32 KBYTES
         L     R6,PAGE8           COUNTER MAXIMUM
         IC    R3,N55             5S USED AS FILLER
LOOP1    STC   R3,0(,R2)          STORE INTO MAP
         LA    R2,1(,R2)          POINTS NEXT BYTE
         LA    R5,1(,R5)          COUNT UP ONE BYTE
         CR    R5,R6              LAST BYTE OF MAP?
         BM    LOOP1              DO AGAIN IF NOT
*
*        INVOKE THE SAVE SERVICE OF THE DIV MACRO
*
         DIV   SAVE,ID=TESTID,SIZE=OBJSIZE
         LTR   R15,R15            CHECK ZERO RC
         BNZ   ERROR              SAVE FAILED
*
*        INVOKE THE UNMAP SERVICE OF THE DIV MACRO
*
         DIV   UNMAP,ID=TESTID,AREA=MAPPTR1
         LTR   R15,R15            CHECK ZERO RC
         BNZ   ERROR              UNMAP FAILED
*
*        OBJECT NOW HAS . CONTIGUOUS PAGES OF 5'S
*
```

The program creates a new window that includes the first eight blocks of the object. This map omits OFFSET, causing a default offset of zero to be used with the specified span of eight blocks. After filling the window with ones, the program invokes RESET against the eighth block of the object which corresponds to the eighth page of the window. Because the information provided by the reset comes from the object which still contains zeroes, the eighth page in the window is set to zeros.

```
*        INVOKE MAP SERVICE FOR FIRST EIGHT 4K
*        BLOCKS OF DATASET, WITH DEFAULT OFFSET.
*
         L     R3,EIGHT           GET VALUE OF SPAN
         ST    R3,SPVALUE         INITIALIZE SPAN
         DIV   MAP,ID=TESTID,AREA=MAPPTR1,              x
               SPAN=SPVALUE
         LTR   R15,R15            CHECK ZERO RC
         BNZ   ERROR              MAP FAILED
*
*        FILL IN DATA - 1'S FOR THE FIRST 8 PAGES
*
         L     R1,MAPPTR1         POINTS TO WINDOW
         LR    R2,R1              POINTS TO MAP
         SR    R5,R5              COUNTER 32 KBYTES
         L     R6,PAGE8           COUNTER MAXIMUM
         IC    R3,N11             1S USED AS FILLER
LOOP2    STC   R3,0(,R2)          STORE INTO MAP
         LA    R2,1(,R2)          POINTS TO NEXT BYTE
         LA    R5,1(,R5)          COUNT UP ONE BYTE
         CR    R5,R6              LAST BYTE OF MAP?
         BM    LOOP2              DO AGAIN IF NOT
*
*     RESET 8TH VIRTUAL BLOCK FROM THE CORRESPONDING
*     BLOCK ON DASD.  THIS BLOCK NOW CONTAINS ZEROES
*     SINCE THE PROGRAM HAS NOT YET INVOKED ANY
*     SAVE SERVICES AFFECTING IT.
*
         L     R3,SEVEN
         ST    R3,OFFS            INITIALIZE OFFSET
         L     R3,ONE
         ST    R3,SPVALUE         INITIALIZE SPAN
         DIV   RESET,ID=TESTID,                         x
               SPAN=SPVALUE,OFFSET=OFFS
         LTR   R15,R15            CHECK IF RC IS ZERO
         BNZ   ERROR              RESET FAILED
```

The program saves the window in the first eight blocks of the object by issuing the DIV macro, specifying SAVE. Then it terminates its use of the object by invoking the UNMAP, UNACCESS, and UNIDENTIFY services of the DIV macro.

```
*        INVOKE SAVE, USING DEFAULTS FOR SPAN AND
*        OFFSET.  THIS SAVES ALL MAPPED BLOCKS ON
*        THE OBJECT.  THE FIRST SEVEN ARE FILLED
*        WITH X'11' AND THE LAST HAS ALL BINARY
*        ZEROES.
*
         DIV   SAVE,ID=TESTID,SIZE=OBJSIZE
         LTR   R15,R15            CHECK ZERO RC
         BNZ   ERROR              SAVE FAILED
*
*        INVOKE THE UNMAP SERVICE
*
         DIV   UNMAP,ID=TESTID,AREA=MAPPTR1
         LTR   R15,R15            CHECK IF RC IS ZERO
         BNZ   ERROR              UNMAP FAILED
*
*        THE OBJECT NOW HAS SEVEN CONTIGUOUS BLOCKS OF
*        1'S, FOLLOWED BY ONE BLOCK  OF 0'S, FOLLOWED BY
*        EIGHT BLOCKS OF 5'S.  NOW INVOKE  UNACCESS.
*
         DIV   UNACCESS,ID=TESTID
         LTR   R15,R15            CHECK IF RC IS ZERO
         BNZ   ERROR              UNACCESS FAILED
*
*        INVOKE THE UNIDENTIFY SERVICE
*
         B     EXIT               SKIP ERROR PROCESSING
ERROR    EQU   *
         L     R15,SIXTEEN        BAD RETURN CODE
         ST    R15,SAVER15        HOLD R15 VALUE
EXIT     EQU   *
         DIV   UNIDENTIFY,ID=TESTID
         LTR   R15,R15            CHECK IF RC IS ZERO
         BZ    FREE               IF SO, LEAVE RC GOOD
         L     R15,SIXTEEN        SET BAD RETURN CODE
         ST    R15,SAVER15        HOLD R15 VALUE
```

Finally, the program frees its virtual storage and goes through a standard exit linkage sequence.

```
*        FREE STORAGE AND EXIT
*
FREE     EQU   *
         FREEMAIN RU,A=MAPPTR1,LV=16*4096,SP=0
         L     R15,SAVER15        RETRIEVE R15
         L     R13,4(R13)         STD EXIT LINKAGE
         L     R14,12(R13)
         LM    R0,R12,20(R13)     SAVE RETURN CODE
         BR    R14
         SPACE 2
*
*        DECLARE VARIABLES
*
MAPPTR1  DS    A             PTR TO GETMAINED STORAGE
OBJSIZE  DS    F             SIZE RETURNED FROM ACCESS
OFFS     DS    A             POSITION IN OBJECT
SPVALUE  DS    A             LENGTH TO BE MAPPED-RESET
SAVER15  DS    F'0'          RC VALUE IN REG 15
SAVEAREA DS    CL72          USED BY DATA-IN-VIRTUAL
TESTID   DS    CL8           ID RETURNED FROM IDENTIFY
DDAREA   DS    CL8
         ORG   DDAREA
         DC    AL1(7)        LENGTH OF DDNAME
         DC    CL7'DYNAMIC'  NAME USED IN JCL
         ORG   DDAREA+8
         SPACE 2
*
*        CONSTANTS
*
N11     DC X'11'             HEX ONES
N55     DC X'55'             HEX FIVES
ONE     DC F'1'              ONE
SEVEN   DC F'7'              SEVEN
EIGHT   DC F'8'              EIGHT
SIXTEEN DC F'16'             SIXTEEN
PAGE8   DC F'32768'          8 TIMES 4096
*
*                    REGISTERS
R0      EQU 0
R1      EQU 1
R2      EQU 2
R3      EQU 3
R5      EQU 5
R6      EQU 6
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
        EJECT
        END   SAMPLE
```

Chapter 14. Sharing application data (name/token callable services)

Where To Find Prerequisite Information

The name/token callable services are available to both unauthorized (problem state and PSW key 8-15) and authorized (supervisor state or PSW key 0-7) programs. *z/OS MVS Programming: Assembler Services Guide* introduces the name/token callable services and discusses functions that are available to all programs. The following topics discuss functions of the name/token callable services that are available to authorized programs only. Before reading the following topics, read about the name/token callable services in *z/OS MVS Programming: Assembler Services Guide*.

Name/token callable services allow you to share data between two programs running under the same task, or between two or more tasks or address spaces. To share data, programs often need to locate data or data structures acquired and built by other programs. The data structures and the programs using them need not reside in the same address space. Name/token callable services provide a way for programs to save and retrieve the data.

Levels for name/token pairs

Name/token pairs have a level attribute associated with them. The level defines the relationship of the creating program (that is, the program that creates the name/token pair) to the retrieving program (that is, the program that retrieves the data). Depending on the level you select, the retrieving program can run under the same task as the creating program, or in the same home address space, in the same primary address space, or in the same system.

- A **task-level name/token pair** allows the creating program and the retrieving program to run under the same task.
- A **home-address-space-level name/token pair** allows the creating program and the retrieving program to run in the same home address space.
- A **primary-address-space-level name/token pair** allows the creating program and the retrieving program to run in the same primary address space.
- A **system-level name/token pair** allows the creating program and the retrieving program to run in the same system. That is, the two programs run in separate address spaces.

The various name/token levels allow for sharing data between programs that run under a single task, between programs that run within an address space, and between programs that run in different address spaces. Some examples of using name/token levels are:

- Different programs that run under the same task can share data through the use of a task-level pair.
- Any number of tasks or SRBs that run within an address space can share data through use of an address-space-level pair.
- Programs that run in different address spaces can share data through the use of a system-level pair.

Determining what your program can do with name/token pairs

The following table shows the name/token callable services your program can use to manipulate different levels of name/token pairs:

Table 36. Summary of What Programs Do with Name/Token Pairs

Service	Unauthorized Programs	Authorized Programs
Create (IEANTCR)	• Task • Home • Primary	• Task • Home • Primary • System
Retrieve (IEANTRT)	Any name/token level	Any name/token level
Delete (IEANTDL)	• Task • Home • Primary **Note:** Unauthorized programs cannot delete any pairs created by authorized programs.	Any name/token level

Note: The task-level and home-level name/token pairs are used by both unauthorized and authorized programs. For complete descriptions of the task-level and home-level name/token pairs, see *z/OS MVS Programming: Assembler Services Guide*.

Checking authorization when retrieving a token

In addition to using the *level* parameter of the IEANTRT callable service to specify the level of a name/token pair, an authorized program can use the parameter to check the authorization of the creator of the pair. Authorized programs might perform such checking to make sure they do not retrieve data from a problem-state and PSW key 8-15 program that was written with an intent to do harm. To retrieve a token and make sure the pair's creating task is supervisor state or PSW key 0-7, use one of the following values on the *level* parameter:

- 11 - task level with authorization check
- 12 - home level with authorization check
- 13 - primary level with authorization check.

If a program requests a retrieval with an authorization check and an unauthorized program created the pair, the program receives a non-zero return code but does not receive the token.

Deciding what name/token level you need

To determine the level to use, consider the relationship between the code that creates the pair and the code that retrieves the pair:

- If the retrieving code will be running under the same task as the creator's code, use the task level
- If the retrieving code will have the same home address space but run under a different task or SRB, use the home level
- If the retrieving code will be running in the same primary address space but run under a different task or SRB, use the primary level

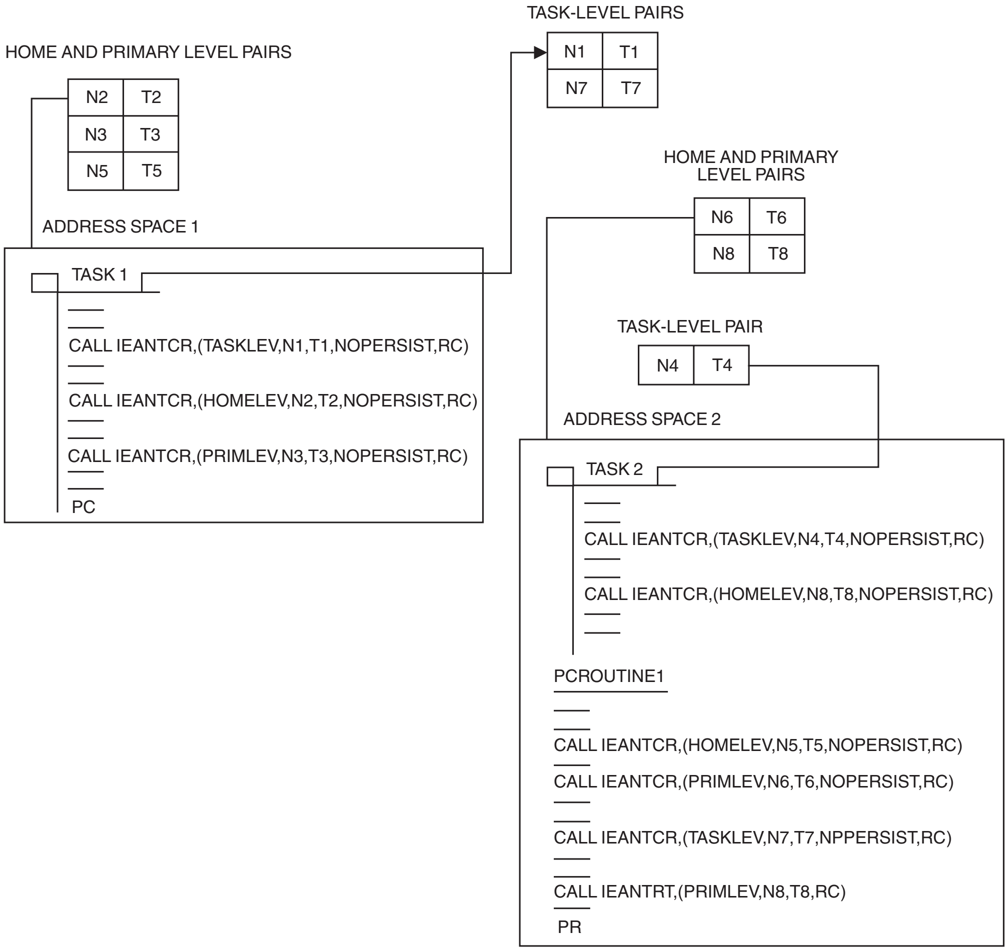
  Note: This level is intended for code running with a primary address space different than the home address space (that is, a cross memory environment).

- If the token will be retrieved by programs in many address spaces, use the system level.

Primary-level name/token pair

A primary-level name/token pair can be used when programs running under different tasks in the primary address space need to use the data.

Figure 51 shows the primary-level name/token pair in a cross-memory environment.



```
CALL IEANTCR,(TASKLEV,N1,T1,NOPERSIST,RC)
CALL IEANTCR,(HOMELEV,N2,T2,NOPERSIST,RC)
CALL IEANTCR,(PRIMLEV,N3,T3,NOPERSIST,RC)
PC
```

```
CALL IEANTCR,(TASKLEV,N4,T4,NOPERSIST,RC)
CALL IEANTCR,(HOMELEV,N8,T8,NOPERSIST,RC)
```

```
PCROUTINE1
CALL IEANTCR,(HOMELEV,N5,T5,NOPERSIST,RC)
CALL IEANTCR,(PRIMLEV,N6,T6,NOPERSIST,RC)
CALL IEANTCR,(TASKLEV,N7,T7,NPPERSIST,RC)
CALL IEANTRT,(PRIMLEV,N8,T8,RC)
PR
```

Figure 51. Using the Primary Level in a Cross-Memory Environment

In Figure 51:

1. TASK 1 creates task-level (N1,T1), home-level (N2,T2), and primary-level (N3,T3) name/token pairs using the IEANTCR callable service.
2. TASK 1 contains a program call (PC) instruction that transfers control to PCROUTINE1 in ADDRESS SPACE 2. PCROUTINE1's home address space is ADDRESS SPACE 1 and its primary address space is ADDRESS SPACE 2.
3. PCROUTINE1 creates a home-level name/token pair (N5,T5) associated with ADDRESS SPACE 1, a primary-level name/token pair (N6,T6) associated with ADDRESS SPACE 2, and a task-level name/token pair (N7,T7) associated with TASK 1 that was dispatched in ADDRESS SPACE 1. PCROUTINE1 also retrieves a primary-level name/token pair (N8,T8) that was previously created by TASK2.

When the primary address space and home address space of the creating task are the same, both the primary-address-space level and the home-address-space level of the pair it creates are the same. For example, when PASN=HASN, a program could create a name/token pair with a primary-address-space level and retrieve it with a home-address-space level. IBM recommends that in this situation, the program use the home-address-space level for retrieving the pair.

When the primary address space and home address space are different, as in a cross-memory environment, the home-address-space level and primary-address-space levels are different. A program can use the home level to access pairs associated with the home address space and use the primary-address-space level to access entries associated with the primary address space. For a given pair, you cannot use the levels interchangeably.

A cross-memory server application might want to retrieve name/token pairs using the primary-address-space level to locate control structures contained in the primary address space. In Figure 51 on page 295, notice that PCROUTINE1 retrieves a primary-level name/token pair (N8,T8). Even though TASK 2 created the name/token pair (N8,T8) as a home-level pair, PCROUTINE1 retrieves the pair as a primary-level name/token pair.

System-level name/token pair

A system-level name/token pair contains data that is shared among programs that run in different address spaces in the same system.

Figure 52 on page 297 shows the system-level name/token pair in a multiple address space environment.

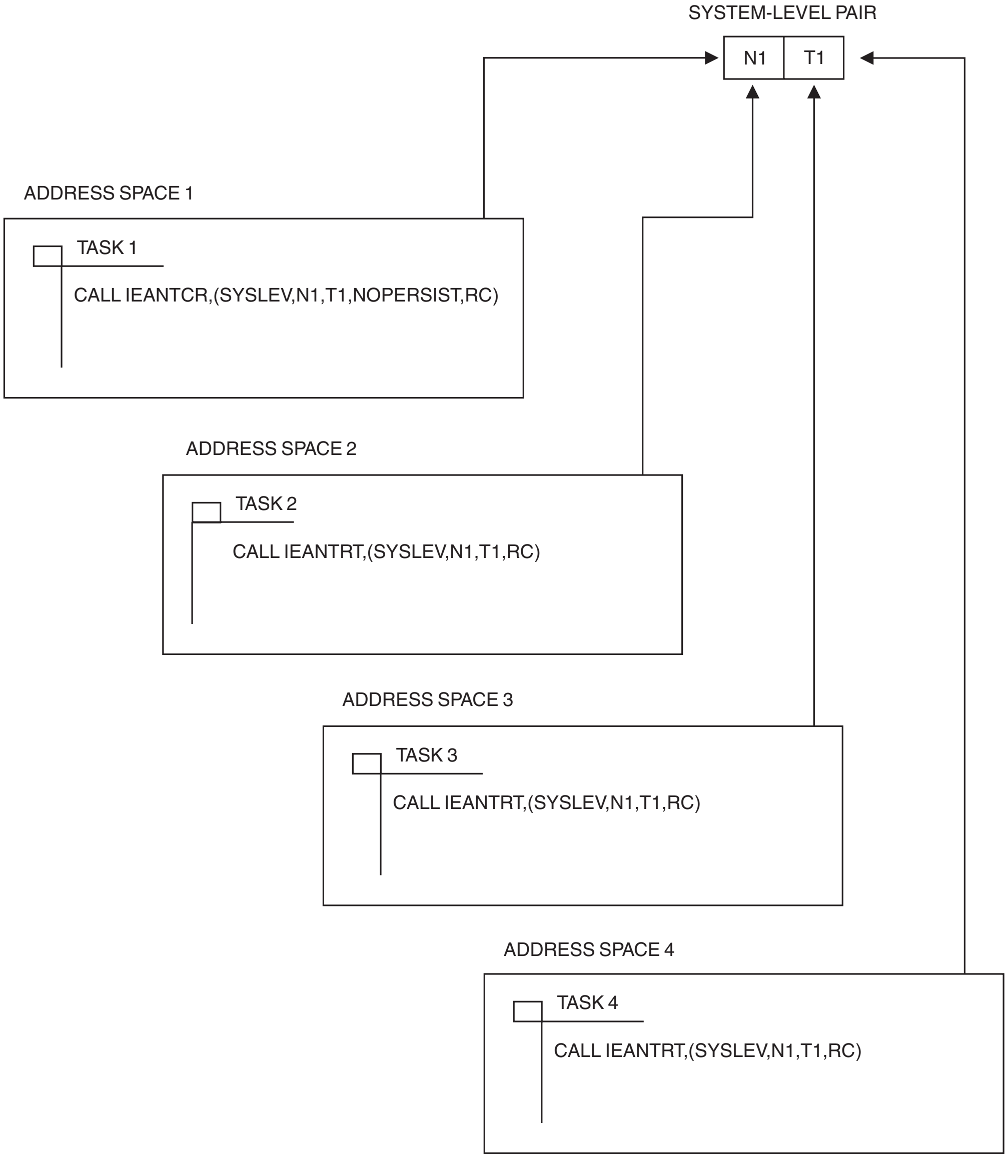

SYSTEM-LEVEL PAIR
N1
T1
ADDRESS SPACE 1
TASK 1
CALL IEANTCR,(SYSLEV,N1,T1,NOPERSIST,RC)
ADDRESS SPACE 2
TASK 2
CALL IEANTRT,(SYSLEV,N1,T1,RC)
ADDRESS SPACE 3
TASK 3
CALL IEANTRT,(SYSLEV,N1,T1,RC)
ADDRESS SPACE 4
TASK 4
CALL IEANTRT,(SYSLEV,N1,T1,RC)

Figure 52. Using the System Level in a Multiple Address Space Environment

In Figure 52:

1. TASK 1 initially creates a system-level name/token pair in ADDRESS SPACE 1 using the IEANTCR callable service.
2. Even though TASKS 2, 3, and 4 reside in different address spaces, they can retrieve the system-level name/token pair using the IEANTRT callable service.

System-level name/token pairs are owned by the job step task of the home address space of the task or SRB that created the pair. System-level pairs are deleted when the owning job step task terminates. However, the IEANTCR callable service provides an option that allows the system-level name/token pair to exist after the owning job step task terminates. This option, available through the **persist_option** parameter on the IEANTCR callable service, is helpful when a program is abnormally terminated and must be restarted. Upon restart, the program can check what data is stored in the name/token pair so the program can free or reuse the program storage.

If the persist option is specified as zero when the system-level name/token pair is created, the pair is deleted when the job step task in the home address space terminates. If the persist option is specified as a one, the pair is not deleted when the job step task in the home address space terminates. However, even if the persist option is specified as a one, the pair will be deleted by the system if the system itself fails and is started again.

If there is not sufficient storage in the system-level name/token pool to permit creation of a new system-level name/token pair, the system obtains additional storage. The system does not release the storage when the pair is deleted. Instead, the system holds the storage for the duration of the IPL, reserving the storage for new system-level name/token pairs. If you create many new system-level name/token pairs without deleting previously-created pairs, a potentially serious storage shortage could result. For this reason, use system-level name/token pairs sparingly.

Owning and deleting name/token pairs

Name/token pairs created by a program are automatically deleted by the system. The level of the name/token pair determines **when** the system deletes the pair.

Note: The words job step in this topic refers to the cross memory resource owning (CMRO) task. While the CMRO task is generally the main job step task, at times it may be either the initiator task or started task control task (such as between jobs).

- Task-level pairs are owned by the task that created them and are deleted when the owning task terminates.
- Home-address-space-level name/token pairs are owned by the job step task of the home address space that created them. These pairs are deleted when the job step task, rather than the creating task, in the address space terminates; that is, home-level pairs created by subtasks of a job step task are not automatically deleted when the subtask terminates.
- Primary-address-space-level name/token pairs are owned by the job step task of the primary address space that created them. These pairs are deleted when the job step task, rather than the creating task, in the address space terminates; that is, primary-level pairs created by subtasks of a job step task are not automatically deleted when the subtask terminates.
- System-level name/token pairs are owned by the job step task of the home address space that created them. If the persist option is specified as zero when the pair is created, the pair is deleted when the owning job step task in the home address space terminates. If the persist parameter is specified as one when the pair is created, the pair is not deleted when the owning job step task in the home address space terminates. The user must explicitly delete the pair when the pair is no longer needed.

  Note: Name/token pairs do not persist when a system terminates and is restarted.

Example of using the name/token services

The following example shows how an application uses the name/token callable services. The application, called a **server application**, offers services to users. The application resides within a single address space, the **server address space**. The application offers services to users that reside in address spaces outside the server address space. The example assumes the server application has already set up the cross-memory environment that allows the users to connect to the server address space.

The server application uses name/token callable service twice during its initialization:

- The first use is to make important information available to the users. Before users can invoke the services of the server application, they need to know the Program Call (PC) number that they use to switch program processing from their primary address spaces to the server's address space. Through IEANTCR, the server, in supervisor state, creates a system-level name/token pair. With a system-level pair, programs running in any address space can retrieve the token, in this case the PC number the user must have to call the server. The invocation that creates the name/token pair is as follows:

  ```
  CALL IEANTCR,(SYSLEV,NAME,TOKEN,NOPERSIST,RETURNCODE)
  ```

  The name/token pair is not marked as persistent; users cannot retrieve the token (that is, the PC number) if the server has terminated. If the server is terminated or in the process of restarting, the user will have to wait until the server is restarted before successfully retrieving the PC number.

  A user retrieves the PC number through the following invocation:

  ```
  CALL IEANTRT,(SYSLEV,NAME,TOKEN,RETURNCODE)
  ```

- The server also uses IEANTCR to create a primary-address-space-level name/token pair. With this name/token pair, any program running in the server's address space can use the name to retrieve the associated token. In this case, the token contains a pointer to the application's main data area. The invocation that creates the name/token pair is as follows:

  ```
  CALL IEANTCR,(PRIMLEV,NAME2,TOKEN2,NOPERSIST,RETURNCODE)
  ```

  If the server abnormally terminates, its recovery routine might use the following invocation to retrieve the address of the main data area. Note that if the recovery routine gets control before the address space terminates, the recovery routine can retry; the name/token pair is still retrievable until the address space completes termination.

  ```
  CALL IEANTRT,(PRIMLEV,NAME2,TOKEN2,RETURNCODE)
  ```

Chapter 15. Processor storage management

The system administers the use of processor storage (that is, central and expanded storage) and directs the movement of virtual storage pages between auxiliary storage slots and real frames in blocks of 4096 bytes. It makes all addressable virtual storage in each address space appear as central storage. Only the virtual pages necessary for program execution are kept in central storage. The remainder reside on auxiliary storage. The system employs the auxiliary storage manager to perform the actual paging I/O necessary to transfer pages in and out of central storage. The system also provides DASD allocation and management for paging space on auxiliary storage.

The system assigns real frames upon request from pools of available frames, thereby associating virtual addresses with central storage addresses. Frames are repossessed when freed by a user, when a user is swapped-out, or when needed to replenish the available pool. While a virtual page occupies a real frame, the page is considered pageable unless it is fixed by the FIX option of the PGSER macro, a PGFIX or PGFIXA macro, or obtained from a fixed subpool. The system also allocates virtual equals real (V=R) regions upon request by those programs that cannot tolerate dynamic relocation. Such a region is allocated contiguously from a predefined area of central storage and is non-pageable.

The PGSER macro provides all the paging services through the use of parameters. PGSER handles virtual addresses above or below 16 megabytes. The macros, PGFIX, PGFIXA, PGFREE, PGFREEA, PGLOAD, PGANY, PGOUT, and PGRLSE are supported to maintain compatibility with MVS/370, but it is recommended that you use the PGSER macro.

The paging services provided include the following:

- Fix virtual storage contents -- PGFIX, PGFIXA, or the FIX option of PGSER
- Fast path to fix virtual storage contents -- the FIX and BRANCH=SPECIAL options of PGSER
- Free central storage -- PGFREE, PGFREEA, or the FREE option of PGSER
- Fast path to free central storage -- the FREE and BRANCH=SPECIAL options of PGSER
- Load virtual storage areas into central storage -- PGLOAD or the LOAD option of PGSER
- Page out virtual storage areas from central storage -- PGOUT or the OUT option of PGSER
- Release virtual storage contents -- PGRLSE or the RELEASE option of PGSER
- Page anywhere (above or below 16 megabytes of central storage) -- PGANY or the ANYWHER option of PGSER
- Makes a range of virtual storage pages read-only -- the PROTECT option of PGSER.
- Makes a range of virtual storage pages modifiable -- the UNPROTECT option of PGSER.

The PGFIX, PGFIXA, PGFREE, and PGFREEA functions as well as the FIX and FREE options of PGSER are available only to authorized system functions and users and are described in the following topics. PGANY, PGLOAD, PGOUT, and

PGRLSE as well as the ANYWHER, LOAD, OUT, and RELEASE options of PGSER are not restricted and are available to all users. PGSER and PGANY are described in this publication. PGLOAD, PGOUT, and PGRLSE are described in *z/OS MVS Programming: Assembler Services Guide* and *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

You can use paging services for storage that you define as shared through the IARVSERV macro. For information about sharing storage through IARVSERV, particularly for the PGSER FIX, PROTECT, and UNPROTECT options, see Chapter 16, “Sharing data in virtual storage (IARVSERV macro),” on page 309.

Fixing/freeing virtual storage contents

Fixing virtual storage and freeing central storage are complementary functions. The PGFIX and PGFIXA macros and the FIX option of PGSER make specified storage areas resident in central storage and ineligible for page-out as long as the requesting address space remains in central storage. Note that page fixing ties up valuable central storage and is usually detrimental to system performance unless the use of the fixed pages is extremely high.

The PGFREE and PGFREEA macros and the FREE option of PGSER make specified storage areas, which were previously fixed via the PGFIX macro or the FIX option of PGSER, eligible for page-out. Pages fixed by PGFIX, PGFIXA, or the FIX option of PGSER are not considered pageable until the same number of page free and page-fix requests have been issued for any virtual area. The fix and free requests for a page must be issued by the same task (unless TCB=0 is specified), otherwise the page will not be freed.

When using the fix function, you have the option of specifying the relative real time duration anticipated for the fix. Specify LONG=Y, if you expect that the duration of the fix will be relatively long. (As a rule of thumb, the duration of a fix is considered long if the interval can be measured on an ordinary timepiece-that is, in seconds.) Additional processing might be required to avoid an assignment of a frame to the V=R area or an area that might be varied offline. Specify LONG=N, if you expect the time duration of the fix to be relatively short. A long-term fix is assumed if you do not specify this option.

In both the fix and free functions, you have the option of specifying that the contents of the virtual area are to remain intact or be released. If the contents are to be released, specify RELEASE=Y; otherwise, specify RELEASE=N. If you specify PGFIX or the FIX option of PGSER with RELEASE=Y, the release function is performed before the fix function. If you specify PGFREE or the FREE option of PGSER with RELEASE=Y, the free function is performed and those pages of the virtual subarea with zero fix counts are released; that is, the contents of virtual areas spanning entire virtual pages that were fixed are expendable and no page-outs for these pages are necessary.

RELEASE may also be specified as a separate parameter of the PGSER macro. When you specify RELEASE as a separate parameter, the released page can be in disabled reference storage.

Note: PGRLSE, PGSER RELEASE, PGSER FREE with RELEASE=Y, and PGFREE RELEASE=Y may ignore some or all of the pages in the input range and will not notify the caller if this was done.

Any pages in the input range that match any of the following conditions will be skipped, and processing continues with the next page in the range:

- Storage is not allocated or all pages in a segment have not yet been referenced.
- Page is in PSA, SQA or LSQA.
- Page is V=R. Effectively, it's fixed.
- Page is in BLDL, (E)PLPA, or (E)MLPA.
- Page has a page fix in progress or a nonzero FIX count.
- Pages with COMMIT in progress or with DISASSOCIATE in progress.

The BRANCH=SPECIAL and the FIX or FREE options of PGSER provide the fast path version of PGSER. The fast path version of PGSER with the FIX option ensure that specific storage areas are resident in central storage and ineligible for page-out. These functions execute only short-term, synchronous page fixes.

Note:

1. PGFIX and the FIX option of PGSER do not prevent pages from being paged out when an entire virtual address space is swapped out of central storage. Consequently, the user of PGFIX and the FIX option of PGSER cannot assume a constant real address mapping for fixed virtual areas in most cases.
2. **IBM recommends that page fixes of more than 100 pages be divided into several smaller fix requests**. Large page fix requests can cause an excessive spin loop to occur.
3. When using the PGFIXA macro or the fast path version of PGSER with the FIX option, or a branch entry to PGSER with the options FIX and TCB=0, fixed areas will not automatically be freed at the end of a job; to free them, issue a PGFREEA macro or the PGSER macro with the FREE and BRANCH=SPECIAL options.

Protecting a range of virtual storage pages

The PROTECT option of PGSER makes a range of virtual storage pages read-only and helps to improve data integrity. The UNPROTECT option of PGSER makes a range of virtual storage pages modifiable and provides a supported way of modifying the PLPA on an IPLed system. Note, however, that PLPA modifications made this way exist only for the life of the IPL.

Areas of storage that can be protected or unprotected above or below 16 megabytes are:

- Private area pages
- SWA pages
- CSA pages
- SQA pages
- LPA pages
- Nucleus pages (except read/only nucleus pages that are backed by 1 megabyte pages)

IBM recommends that you use PGSER PROTECT only for full pages of storage on page boundaries. This usage avoids making other areas of storage read-only unintentionally. For instance, if you obtain a virtual storage area smaller than a page and then issue PGSER PROTECT, the entire page is made read-only, including the portion of the page that is not part of your storage area.

PROTECT and UNPROTECT apply only to virtual storage pages. The system does not keep track of how many times a page has been protected or unprotected. One UNPROTECT cancels all PROTECTs for a given page.

PGFIX/PGFREE completion considerations

Under normal circumstances, you can reverse the effect of a PGFIX by using a PGFREE when the need for a page fix ceases. You can also reverse the effect of the FIX option of PGSER by using the FREE option of PGSER when the need for a page fix ceases. However, a page-fix request sometimes completes asynchronously if, for example, it requires a page-in operation. In such cases, you might need to explicitly purge page-fix operations.

For this reason, the page-fix function provides a mechanism for signalling event completion. The mechanism is the standard ECB together with WAIT/POST logic. The requestor supplies an ECB address and waits on the ECB after a request if the return code indicates that all of the pages were not immediately fixed. The ECB is posted when all requested pages are fixed in central storage.

Note: Callers who supply an ECB and use PGSER must check the return code before waiting since the ECB is not posted for a return code of 0.

There are two ways to explicitly purge a page fix:

- If the page fix is known to be complete, the page fix is reversed through the page-free function.
- If there is any possibility that the page fix has not been posted as complete, issue PGFREE or PGSER with FREE and supply an ECB address. This ECB parameter identifies the event control block that was supplied as an input parameter with the page fix being purged. Note that for the purpose of canceling a page-fix request that has not yet completed, the ECB must uniquely identify the page-fix request. Consequently, to provide for explicit purging, you must ensure that the ECB for any incomplete page fix can be located in a purge situation, and that the ECB has not been reused at the time the page fix is to be canceled.

The page-free function always completes immediately and requires no ECB address except for purging considerations.

The issuer of the following macros is responsible for freeing the fixed frames:

- PGFIXA
- PGSER, with the FIX, BRANCH, and TCB=0 options
- PGSER, with the FIX and BRANCH=SPECIAL options

This can be accomplished by using PGFREEA; PGSER with FREE, BRANCH, and TCB=0; or PGSER with FREE and BRANCH=SPECIAL.

An FRR (functional recovery routine) or ESTAE recovery routine should be established during the period these fixes are outstanding. The recovery routine should free the frames in case there is an unexpected error.

Input to page services

There are two formats for providing input to page services. These are the register (R) and list (L) formats. If you specify R, page services uses the input information supplied in registers to perform the requested function; if you specify L, page services uses the input information provided in a parameter list to perform the requested function. The information that you must provide in the parameter list includes the starting and the ending addresses for which you want the page service to be performed and an indication of the end of the list.

The list used depends on which page services macro you code. Descriptions of the parameter lists and the macros that use them follow.

Virtual subarea list (VSL)

The virtual subarea list provides the basic input to the page service functions: PGFIX, PGFIXA, PGFREE, PGFREEA, PGLOAD, PGRLSE, and PGOUT.

The list contains one or more doubleword entries; each entry describes an area in virtual storage. The list must be non-pageable and located in the address space to be processed. The VSL is not required to be on a word boundary.

For a description of the VSL, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Page service list (PSL)

The page service list provides the basic input to the page service functions of the PGSER macro with the exception of the BRANCH=SPECIAL option. Each entry in the list specifies a range of addresses to be processed, or specifies the address of the next list entry to be processed, or is null. The first entry also indicates the paging service that is to be performed on all the ranges specified in the list.

The PSL has the following characteristics:

- The list must be in non-pageable storage.
- The PSL is not required to be on a word boundary.
- All addresses specified are 31-bit addresses.

For a description of the PSL, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Short page service list (SSL)

The short page service list provides the basic input to the PGSER macro with the BRANCH=SPECIAL option. The list contains entries for the 31-bit starting and 31-bit ending addresses of the virtual area to be fixed or freed.

The SSL has the following characteristics:

- The list must be in non-pageable storage.
- The SSL is not required to be on a word boundary.
- All addresses specified are 31-bit addresses.

For a description of the SSL, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Branch entry to page services

Branch entry is available for all page services, but macro interfaces are provided only for the PGSER, PGFIXA, and PGFREEA macros. The PGSER macro, which is the IBM-recommended interface to page services, supports branch entry in both cross memory and non-cross memory mode. The PGFIXA and PGFREEA macros also work in cross memory and non-cross memory mode but should be used only under the special circumstances for which they are intended. The page-fix, page-free, page-load, page-release, page-any, and page-out services support both cross memory and non-cross memory mode; the page-out service supports only non-cross memory mode.

To branch-enter page services that do not provide a macro interface for branch entry, the caller must:

- Run in supervisor state with PSW key 0.
- Hold the local lock of the currently addressable address space.
- Set up the registers as described below for cross memory or non-cross memory mode.
- Branch to the service.

Note: The pages that are candidates for page services must be addressable in the current address space. All addresses must be 24-bit addresses.

Cross memory mode

Register(s)	Bit(s)	Contents
0		0
1	0	0 for register format 1 for list format
	1-7	Same as bits 1-7 of VLSFLAG1 field of VSL for register format; irrelevant for list format
	8-31	24-bit starting address on which the service is to be performed for register format; 24-bit address of user's first VSL for list format
2	0-7	Same as VSLFLAG2 field in VSL for register format; irrelevant for list format
	8-31	24-bit ending address + 1 for which the service is to be performed for register format; irrelevant for list format
3		Irrelevant
4		0
5-6		Irrelevant
7		ASCB address of current address space
8-13		Irrelevant
14		Return address
15		Entry point to page services, obtained from the CVT field, CVTPSXM.

On return, the page service sets the registers as follows:

Register
Contents

0-4 Used by the system as work registers.

5-14 Unchanged.

15 Return code.

The only return code possible is 0. This indicates that the requested function was processed successfully.

Non-cross memory mode

The caller must set up registers as follows:

Register(s)	Bit(s)	Contents
0		ECB address or 0 if no ECB is specified
1	0	0 for register format 1 for list format
	1-7	Same as bits 1-7 of VSLFLAG1 field of VSL for register format; irrelevant for list format
	8-31	24-bit starting address on which the service is to be performed for register format; 24-bit address of user's first VSL for list format
2	0-7	Same as VSLFLAG2 field in VSL for register format; irrelevant for list format
	8-31	24-bit ending address + 1 for which the service is to be performed for register format; irrelevant for list format
3		Irrelevant
4		TCB address or 0
5-13		Irrelevant
14		Return address
15		Entry point to page services, obtained from the CVT field, CVTVPSIB.

On return, the page service sets the registers as follows:

Register
Contents

0-14 Unchanged

15 Return code

The return codes are as follows:

Code **Meaning**

0 The requested function was processed successfully. If the function was page-fix or page-load, and an ECB was supplied, it will be posted.

8 The requested function was page-fix or page-load with an ECB. The function will be processed asynchronously and the ECB will be posted upon completion.

12 The requested function was page-out and the function was unsuccessful for at least one of the specified pages.

Chapter 16. Sharing data in virtual storage (IARVSERV macro)

With the shared pages function, which is available through the IARVSERV macro, you can define virtual storage areas through which data can be shared by programs within or between address spaces or data spaces. Also, the type of storage access can be changed.

Sharing reduces the amount of processor storage required and the I/O necessary to support data applications that require access to the same data. For example, IARVSERV provides a way for a program running below 16 megabytes, in 24-bit addressing mode, to access data above 16 megabytes that it shares with 31-bit mode programs. IARVSERV allows the sharing of data without the central storage constraints and processor overhead of other existing methods of sharing data.

The sharing of data benefits many types of applications, because data is available to all sharing applications with no increase in storage usage. This function is useful for applications in either a sysplex environment or a single-system environment. Additionally, IARVSERV allows you to control whether a sharing program:

- Has read access only
- Has both read and write access and receives updates immediately
- Can modify the data without modifying the original, and without allowing the sharing programs to view the updates
- Can modify the original while sharing programs see the change, but without allowing the sharing programs to change the data
- Can change the current storage access

An additional macro, IARR2V, is provided as an aid to converting central storage addresses to virtual storage addresses. See “Converting a central to virtual storage address (IARR2V macro)” on page 318 for information on the IARR2V macro.

The IARVSERV topics described in this information are:

- Understanding the concepts of sharing data with IARVSERV
- Storage you can use with IARVSERV
- Obtaining storage for the source and target
- Defining storage for sharing data and access
- Changing storage access
- How to share and unshare data
- Accessing data in a sharing group
- Example of sharing storage with IARVSERV
- Use with data-in-virtual (DIV macro)
- Use with Page Services (PGSER macro)
- Diagnosing problems with shared data

For coding information about the IARVSERV and IARR2V macros, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Understanding the concepts of sharing data with IARVSERV

As you read this information, refer to Figure 53 for an illustration of the sharing data through the IARVSERV macro.

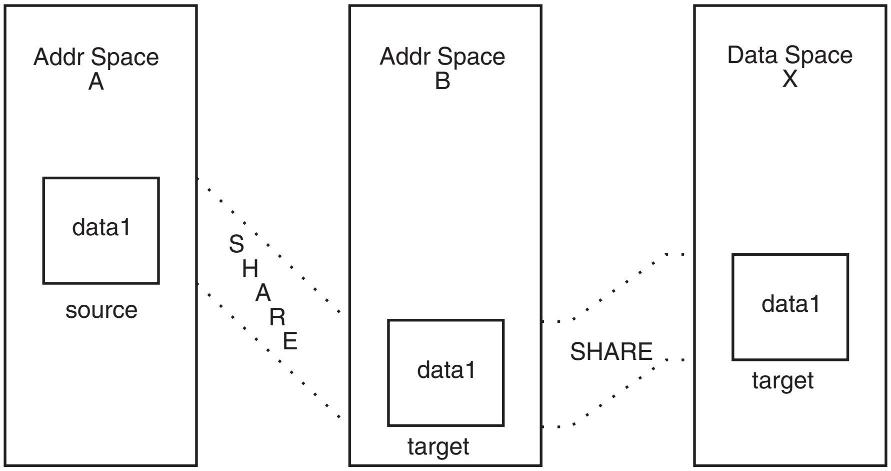


Figure 53. Data Sharing with IARVSERV

Suppose that Addr Space A contains data that is required by programs in Addr Space B. A program in Addr Space A can use IARVSERV to define that data to be shared; that data and the storage it resides in are called the **source**. The program also defines storage in Addr Space B to receive a copy of the source; that storage and its copy of source data are called the **target**.

The source and its corresponding target form a **sharing group**. A sharing group can consist of several target areas and one source. For example, suppose another program in Addr Space A defines a portion of data1 (in Addr Space A) as source, and defines a target in Data Space X. That target becomes a member of the sharing group established previously.

All sharing of data is done on a page (4K) basis. If the source page is already a member of an existing sharing group, the target becomes a member of that existing sharing group. A page is called a **sharing page** if it is a member of a sharing group.

Programs that access the source or targets are called **sharing programs**. Each sharing program accesses the shared virtual storage as it would any other storage, and may not need to know that the storage is being shared. So, you can allow programs to share data through IARVSERV without having to rewrite existing programs.

Storage you can use with IARVSERV

You can share data in address spaces and data spaces. You can use any storage to which you have valid access, except for a subspace, a hiperspace, a VIO window, a V=R region, the PSA or the nucleus (read-only, extended read-only, read-write and extended read-write areas). The target area cannot contain page-protected or page-fixed pages. Storage that is currently allocated through DIV MAP to any DIV object cannot be the target, nor can the source storage have been allocated through DIV MAP to a hiperspace.

Authorized programs can use an unlimited number of pages for sharing. However, use care to ensure that it is used appropriately. System performance can be affected if programs use large amounts of storage for many source areas, but having many targets of one large or a few small sources can reduce central storage constraints.

ESQA storage requirements increase as more storage is shared. The formula for the number of bytes of ESQA storage used by the systems is: 40 bytes per shared page group plus 40 bytes per shared page view. For example, a single shared page group with two views requires 40+40*2 = 120 bytes of ESQA storage.

The maximum number of shared pages for a program in problem state with PSW key 8-15 is 32, unless this number is modified by your installation. This number includes both the source and targets, so the actual number of unique pages is 16. Your installation can use the IEFUSI installation exit to change the limit for problem state programs. See *z/OS MVS Installation Exits* for details on IEFUSI.

In order to expedite the return of all internal control blocks for the shared storage back to the system, IBM recommends issuing IARVSERV UNSHARE against all views for both source and target that are originally shared. For an example of how to code the UNSHARE parameter, see *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

Obtaining storage for the source and target

Before you can issue IARVSERV to define storage as shared, you must obtain or create both the source and target areas. For address space storage, use the GETMAIN or STORAGE macro; for data space storage, use the DSPSERV macro. The source and target areas must be as follows:

- Start on a page boundary,
- Have the same storage protect key and fetch-protection status (except for TARGET_VIEW=UNIQUEWRITE or TARGET_VIEW=LIKESOURCE and the source has UNIQUEWRITE view),
- Meet one of the following requirements:
  - Reside within pageable private storage of an address space.
  - Reside within the valid size of an existing data space and be pageable storage.

The source can be common storage and the target can be in CSA/ECSA, SQA/ESQA, LSQA/ELSQA or DREF storage. The target area must not be storage that contains system data, including storage containing access lists and dynamic address translation tables. The source and the target must be two different storage areas. They must be different virtual storage addresses or reside in different address or data spaces.

Then initialize the source with data. Make sure any storage you obtain or data space you create can be accessed by the intended sharing programs. For example, if you want to allow sharing programs to both read and modify a target, the programs' PSW key value must match or override the target's storage protection key. For information about storage subpool attributes, see Table 32 on page 228. For information about access to data spaces, see *z/OS MVS Programming: Extended Addressability Guide*.

Defining storage for sharing data and access

With the IARVSERV macro, you can define multiple types of data sharing and access. As you read this information, use Figure 53 on page 310 to see how each IARVSERV parameter acts on the current state of the data. Each type of data sharing access is called a specific **view** of the source data. A view is the way your program accesses, or sees, the data. You define the view in the TARGET_VIEW parameter on IARVSERV, by specifying one of the following:

- Read-only view (READONLY value) — where the target data may not be modified.
- Shared-write view (SHAREDWRITE value) — where the target data can be read and modified through the view.
- Copy-on-write view (UNIQUEWRITE value) — where the source data modifications are not seen by other source - sharing programs. Any attempt to modify the shared source data in this view causes MVS to create a unique target copy of the affected page for that address or data space.

  An example of two different cases:

  - If the shared data is modified through a SHAREDWRITE view, the UNIQUEWRITE view gets an unmodified copy of the data. Any remaining views sharing that data see the modified data.
  - If the shared data is modified through a UNIQUEWRITE view, the UNIQUEWRITE view gets the modified copy of the data. Any remaining views sharing that data see the unmodified data.
- Copy-on-write target view (TARGETWRITE value) — where the target data may be read and modified through the source view. Any modification of a shared target page causes MVS to create a unique target copy of the affected page for that address or data space.

  An example for two different cases:

  - If the shared data is modified through a SHAREDWRITE view, the TARGETWRITE view sees the modified data.
  - If the shared data is modified through a TARGETWRITE view, the TARGETWRITE view sees the modified copy of the data. Any remaining views sharing that data see the unmodified data.
- Like source view (LIKESOURCE value) — where the target data is given the current view type of the source data. If the source data is currently not shared, then its current storage attribute is given to the target.
- Hidden view (HIDDEN value) — where the target will share the source data, but any attempt to access the target data (HIDDEN value) will cause a program check. To access the target, the view type must be changed to READONLY, SHAREDWRITE, UNIQUEWRITE, or TARGETWRITE.

When you specify a value for TARGET_VIEW, keep the following in mind:

- The execution key (PSW key) of the caller must be sufficient for altering the target area. If TARGET_VIEW=SHAREDWRITE is specified, the execution key must be sufficient for altering the source area also.
- The SHAREDWRITE value cannot be specified if the source contains sharing pages that have the READONLY view mode attribute, are page protected, or are protected by attribute (such as PLPA).
- For TARGET_VIEW=UNIQUEWRITE, if the input source area is address space storage, and the storage has not been obtained by GETMAIN or STORAGE OBTAIN, or the storage and fetch protect keys do not match, then the SHARE is

not performed for that area. The target will be all zeros (first reference), or it will remain as pages that were not obtained by GETMAIN.

- For target views created with LIKESOURCE on IARVSERV SHARE, the system propagates explicit page protection from the source to the target view.
- Page-fixed pages and DREF pages cannot be made TARGETWRITE, UNIQUEWRITE, or HIDDEN.

Changing storage access

With the IARVSERV macro, the SHARE and CHANGEACCESS parameters can change the views type of storage access. For SHARE, the current storage attribute of the source data affects the outcome of the target. Table 37 shows the permitted target views for different combinations with the source. A NO in the table means that an abend will occur if you request that target view with the current source view. For CHANGEACCESS, all combinations are permitted.

Table 37. Allowed Source/Target View Combinations for Share (Requested Target View)

Current Source View	READONLY	SHAREDWRITE	UNIQUEWRITE	TARGETWRITE	HIDDEN	LIKESOURCE
READONLY	Yes	No	Yes	Yes	Yes	Yes
SHAREDWRITE	Yes	Yes	Yes	Yes	Yes	Yes
UNIQUEWRITE	Yes	Yes	Yes	Yes	Yes	Yes
TARGETWRITE	No	No	Yes	No	No	Yes
HIDDEN (Shared)	No	No	No	No	No	Yes
Non-Shared	Yes	Yes	Yes	Yes	Yes	Yes
HIDDEN (Non-Shared)	No	No	No	No	No	Yes

The following apply when using IARVSERV SHARE when changing storage access:

- For source views to be either UNIQUEWRITE or TARGETWRITE, the processor must have the Suppression-On-Protection (SOP) hardware feature, and a previous IARVSERV SHARE must have created a view of UNIQUEWRITE or TARGETWRITE.
- For target views to be TARGETWRITE, the processor must have the SOP hardware feature. If a request is made to create a TARGETWRITE view and the SOP feature is not installed, the request fails with a return code of 8.
- For target views to be UNIQUEWRITE, the SOP hardware feature must be installed. Also, the request must not specify COPYNOW. If the request specifies COPYNOW, or the SOP feature is not installed, a UNIQUEWRITE view is not established, and a separate copy of the data is made.
- For target views created with LIKESOURCE on IARVSERV SHARE, the system propagates explicit page protection from the source to the target view.
- For source pages that are not shared, if the page is page-protected, the view created for that page is a SHAREDWRITE view, but the view is flagged as an explicitly protected view (one that cannot be modified).

The following apply when changing the storage access with IARVSERV CHANGEACCESS:

- To remove hidden status, you must use an IARVSERV CHANGEACCESS, FREEMAIN, or DSPSERV DELETE macro.

- To remove explicit read-only protection status, you must use an IARVSERV CHANGEACCESS, FREEMAIN, DSPSERV DELETE, or PGSER UNPROTECT macro.
- If a hidden page is hidden because of loss of access to 'mapped' data (such as through DIV UNMAP), and, if the page is changed from hidden, the data in the page might be lost.
- Hidden pages cannot be released via a PGSER RELEASE or DSPSERV RELEASE macro. An attempt would result in an abend with the same reason code as is used for protected pages.
- Issuing an IARVSERV UNSHARE macro for the original mapped page causes the data to be retained for that page. The data for the other sharing pages is lost. References to hidden pages cause an X'0C4' abend, and references to lost pages cause in a X'028' abend.
- Page-fixed pages and DREF pages cannot be made TARGETWRITE, UNIQUEWRITE, or HIDDEN.

How to share and unshare data

With the IARVSERV macro, use the SHARE parameter to initiate sharing of data; use the UNSHARE parameter to end sharing for the issuing program. This information discusses the additional IARVSERV parameters that you can specify with SHARE or UNSHARE.

The **RANGLIST** parameter is always required for both SHARE and UNSHARE. It gives IARVSERV information about the source and target addresses. The RANGLIST value is actually the address of the list of addresses you must create using the mapping macro IARVRL. For the details of IARVRL, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). The following table lists the required IARVRL fields that you must supply for SHARE or UNSHARE.

IARVRL Fields That You Must Initialize for SHARE	IARVRL Fields That You Must Initialize for UNSHARE
VRLSVSA VRLSSTKN (for STOKEN) VRLSALET (for ALET) VRLNUMPG VRLTVSA VRLTSTKN (for STOKEN) VRLTALET (for ALET)	VRLNUMPG VRLTVSA VRLTSTKN (for STOKEN) VRLTAKET (for ALET)

For IARVSERV SHARE, if the target area contains pages that belong to an existing sharing group, MVS performs an implicit UNSHARE to pull those pages out of the existing sharing group before proceeding. Also, MVS automatically performs an UNSHARE on any sharing page when the page is being freed by FREEMAIN, STORAGE RELEASE, or DSPSERV DELETE, or when the page's address space is ended.

Also, when MVS finds that one page of a range is not acceptable for sharing, MVS will not complete the SHARE request for that page, nor the rest of the range or ranges not already processed. You can assume that all pages up to that point were processed successfully. An abend will be issued and GPR 2 and 3 will contain the address range list associated with the error page and the storage address of the page in error, respectively. To remove the SHARE on the successful pages, issue IARVSERV UNSHARE for the storage ranges up to, but excluding, the error page.

The parameter **TARGET_VIEW** is required with SHARE only, to tell IARVSERV how you plan to share the data contained in the source. You have three choices described in "Defining storage for sharing data and access" on page 312.

- READONLY does not allow any program accessing the target area to write to it. An abend results if a program attempts to write to a READONLY target.
- SHAREDWRITE allows any sharing program to write to the target. All those sharing the target area instantly receive the updates. This view could be very useful as a communication method for programs.
- UNIQUEWRITE has the property of copy-on-write, which means that MVS creates a copy of a page for the updating program once the program writes to that page. The only program that has the change is the program that changed it; all others continue to use the original page unmodified. This is true whether the program writes to a source or target page.

  A copy-on-write hardware facility is provided for additional performance improvement. If you need to determine if your processor has the feature, you can use the CVT mapping macro, and test the CVTSOPF bit. For details on the CVT mapping macro, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

RETAIN is a parameter available only with UNSHARE. RETAIN=NO requests that MVS remove the target from sharing. The target data is lost. RETAIN=YES requests that MVS leave the data in the target untouched.

Specifying RETAIN=NO for an unshare of a E/LSQA or E/SQA view will result in a page that is not backed by real storage. Any subsequent access to the page will result in an 0C4 abend. The page can, however, be the target of a subsequent share. Specifying RETAIN=YES for an unshare of a E/LSQA or E/SQA page will result in a 6C5 abend.

Accessing data in a sharing group

Data is accessed in a sharing group just as it would be if sharing did not exist. Trying to write to a READONLY view will cause an abend.

You can create a sharing group that permits programs in 24-bit addressing mode to access data above 16 megabytes. To do this, you would define the source in storage above 16 megabytes, and obtain a target in storage below 16 megabytes. Then initialize the source with data, so programs in 24-bit mode can share the data through the target.

Example of sharing storage with IARVSERV

Suppose you are updating a program called PGMA, that controls all the account deposits for a savings bank. Your program must work with two older programs that are complex and do not have source code available. The first program, called SUBPGMA, was updated six years ago and runs in 31-bit addressing mode; it records deposits in money market accounts. It cannot use data spaces. The other program, SUBPGMB, is much older and records deposits in standard savings accounts. It runs in 24-bit addressing mode. See Figure 54 on page 316 for a representation of the storage.

Program PGMA, the main program, was written to keep all of its data in one large data space. PGMA must continually obtain appropriate storage in the address space that is addressed by SUBPGMA and SUBPGMB. After SUBPGMA and SUBPGMB finish, PGMA must copy all the updated data back to the data space. This is degrading performance and needs to be fixed. By using IARVSERV, you can eliminate the copying, and reduce the complexity of PGMA.

Your update to PGMA would cause the programs to work together this way:

1. PGMA creates a data space and initializes it with data.
2. Before PGMA calls SUBPGMA to do a money market deposit, PGMA issues GETMAIN for storage in the private area for a buffer. This buffer is BUFFER31.
3. PGMA issues IARVSERV SHARE to share the source in the data space with the target, BUFFER31. Use TARGET_VIEW=SHAREDWRITE so updates can be made directly into the data space.
4. PGMA now calls SUBPGMA to update the data, passing the address of BUFFER31 as the area to be updated.
5. Once SUBPGMA updates the data in BUFFER31, PGMA issues IARVSERV UNSHARE followed by FREEMAIN to release the storage.
6. When PGMA needs to call SUBPGMB to do a savings account deposit, the only difference is that PGMA must obtain storage below 16 megabytes for the buffer. This buffer is BUFFER24.
7. PGMA again issues IARVSERV SHARE with TARGET_VIEW=SHAREDWRITE, but identifies the target as BUFFER24.
8. PGMA calls SUBPGMB to update the data, passing the address of BUFFER24 as the area to be updated.
9. Once SUBPGMB updates the data in BUFFER24, PGMA issues IARVSERV UNSHARE and FREEMAIN to release the storage as before.

Note that all three programs could share the data in the data space at the same time. Sharing continues until PGMA issues IARVSERV UNSHARE for that buffer area.

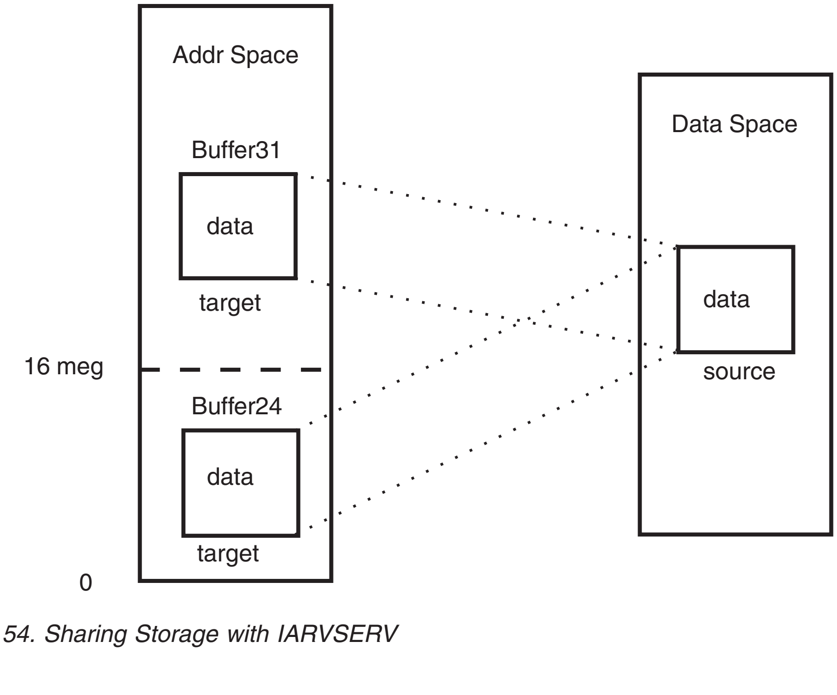


Figure 54. Sharing Storage with IARVSERV

Use with data-in-virtual (DIV macro)

There are several restrictions for programs that use data-in-virtual MAP service with data shared using the IARVSERV SHARE service:

- A sharing page must reside in non-swappable storage and have a SHAREDWRITE view mode.
- Only one member of a sharing group can be mapped. Any attempt to map another member of the same sharing group results in a X'08B' abend.
- You cannot use the IARVSERV macro to share data mapped to a hiperspace object.
- You cannot map a sharing page whose sharing group contains a page that is currently a fixed or disabled reference page.
- If the owning address space of a sharing page that was mapped by DIV MAP terminates prior to a DIV UNMAP, the data is lost. Any further reference to the shared data results in a X'028' abend.

There are also restrictions for programs that use the data-in-virtual UNMAP function:

If a sharing page is currently mapped, and the owner of the map issues DIV UNMAP with RETAIN for that page, the value of RETAIN affects all sharing group members as follows:

- For RETAIN=NO, all pages of the target become unpredictable in content.
- For RETAIN=YES, all pages of the target get the data as it last appeared within the sharing page. This can be useful for saving an instance of data, such as a check point. Use of RETAIN=YES can affect performance if it consumes large amounts of central storage by repeated retaining of the storage.

Use with page services (PGSER macro)

If programs are sharing data using the IARVSERV macro, the highest level of protection remains in effect to ensure data integrity. For example, pages that have been shared with a READONLY view, and are then protected through PGSER, will remain READONLY protected even after PGSER is issued with UNPROTECT. Likewise, pages that have been shared with a READONLY view, and are protected through PGSER, will remain protected even after IARVSERV is issued with UNSHARE.

Diagnosing problems with shared data

You can use IPCS reports to see how data is being shared through IARVSERV. The IPCS RSMDATA subcommand with the SHRDATA parameter provides a detailed report on the status of IARVSERV data sharing. The following RSMDATA reports also provide shared data information: ADDRSPACE, EXPFRAME, REALFRAME, RSMREQ, SUMMARY, and VIRTPAGE. See *z/OS MVS IPCS Commands* for more information about the SHRDATA subcommand.

You may also collect information about data shared through IARVSERV by issuing the DISPLAY command, and by specifying certain optional parameters on the IARR2V macro. See *z/OS MVS System Commands* and “Converting a central to virtual storage address (IARR2V macro)” on page 318 for more information.

Converting a central to virtual storage address (IARR2V macro)

The IARR2V macro provides a simple method to obtain a virtual storage address from a central storage address. This conversion can be useful, for example, when you are working with an I/O or diagnostic program that provides central storage addresses, but you want to use virtual storage addresses.

The details of the syntax and parameters of IARR2V are in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*. In its simplest form, the IARR2V macro requires only the RSA parameter. The RSA parameter specifies the central storage address that you want to convert.

The system returns the virtual storage address in a register or in a storage location you specify through the VSA parameter. Also, you can request the system to return the ASID or STOKEN of the address space or data space associated with the address. For an authorized program, you may also specify the type of linkage the system is to use.

If you require knowledge of whether the central storage address you have is being shared through the IARVSERV macro, you can get that information using the WORKREG, NUMVIEW, and NUMVALID parameters. To use the NUMVIEW and NUMVALID parameters, you must use the WORKREG parameter to specify the work register for the system to use. The NUMVIEW parameter requests the total number of pages sharing the view of your central storage address. NUMVALID requests the number of pages currently addressable (accessed), which is called the number of valid views. With NUMVIEW and NUMVALID, you can check how effectively programs are using shared storage. Pages that are not accessed have not been read or updated by any program.

Chapter 17. The nucleus

The nucleus contains routines that execute with dynamic address translation (DAT) turned off and routines that execute with DAT on.

There are three macros that provide services for the nucleus. These macros are:

Macro **Function**

DATOFF
Provides a means of linking to routines in the DAT-OFF nucleus

NUCLKUP
Provides a means of obtaining information about CSECTs in the DAT-ON nucleus

NMLDEF
Provides a means of identifying the members in SYS1.NUCLEUS that are to be loaded into the DAT-ON nucleus region.

Linking to routines in the DAT-OFF nucleus (DATOFF)

The DAT-OFF nucleus is not mapped in virtual storage. IPL processing loads the DAT-OFF nucleus into consecutive real storage located at the highest available real address. Because the DAT-OFF nucleus is not mapped in virtual storage, a special method is used to link to routines in this area. The DATOFF macro provides the means of linking to system and user-written routines in the DAT-OFF nucleus.

When using the DATOFF macro, the caller specifies an index that identifies the routine that is to receive control in the DAT-OFF nucleus. The index, entry point, and purpose of the routines available to users in the DAT-OFF nucleus are shown in Table 38.

Table 38. DAT-OFF Routines Available to Users

Index	Entry Point	Function
INDCDS		31-bit Compare Double and Swap
INDMVCL0		31-bit General DAT-OFF MVCL
INDMVCLK		31-bit General DAT-OFF MVCL in user key
INDXC0		31-bit General DAT-OFF XC
INDCDS64		64-bit Compare Double and Swap
INDMVCL64		64-bit General DAT-OFF MVCL
INDMVCLK64		64-bit General DAT-OFF MVCL in user key
INDXC64		64-bit General DAT-OFF XC
INDUSR1	IEAVEUR1	31-bit user defined
INDUSR2	IEAVEUR2	31-bit user defined
INDUSR3	IEAVEUR3	31-bit user defined
INDUSR4	IEAVEUR4	31-bit user defined
INDUSR641	IEAV64U1	64-bit user defined
INDUSR642	IEAV64U2	64-bit user defined
INDUSR643	IEAV64U3	64-bit user defined
INDUSR644	IEAV64U4	64-bit user defined

All routines that execute with DAT turned off must be located in the DAT-OFF nucleus. These routines receive control and execute in 31-bit addressing mode or 64-bit addressing mode and must be capable of residing either above or below 16 megabytes. Therefore routines that execute in the DAT-OFF nucleus must have the attributes AMODE=31 or AMODE=64, RMODE=ANY. For information concerning 24-bit/31-bit compatibility, see *z/OS MVS Programming: Assembler Services Guide*.

Using system provided DAT-OFF routines (DATOFF)

The system defined index values, INDMVCL0, INDMVCLK, INDXC0, and INDCDS, and their 64-bit counterparts, are available to users. INDMVCL0 initiates the move character long (MVCL) function, INDMVCLK initiates the MVCL function in user key, INDCDS initiates the compare double and swap function, and INDXC0 initiates the exclusive OR (XC) function. The register usage and linkage for these functions follows.

In all cases, the DATOFF macro destroys the contents of general registers 0, 14, and 15.

INDMVCL0 and INDMVCL64 - Move character long

INDMCVL0 input register values must be 31-bit addresses. INDMCVL64 input register values must be 64-bit addresses. Before issuing the macro, the user must load the registers as follows:

Register	**Use**
0	Used as a work register by the system
2	Real location into which the characters are to be moved
3	Length of the area into which the characters are to be moved
4	Real location of the area from which the characters are to be moved
5	Length of the area from which the characters are to be moved
14	Used as a work register by the system
15	Used as a work register by the system
1,6-13	Unused

The user invokes the 31-bit MVCL function by coding the following macro:

```
DATOFF INDMVCL0
```

INDMVCLK and INDMVCL64 - Move character long in user key

INDMVCLK input register values must be 31-bit addresses. INDMVCLK64 input register values must be 64-bit addresses. Before issuing the macro, the user must load the registers as follows:

Register	**Use**
0	Used as a work register by the system
2	Real location into which the characters are to be moved
3	Length of the area into which the characters are to be moved
4	Real location of the area from which the characters are to be moved
5	Length of the area from which the characters are to be moved

6 Bits 24-27 contain the PSW key in which the MVCL function is to be performed

14 Used as a work register by the system

15 Used as a work register by the system

1,7-13 Unused

The user invokes the 31-bit MVCL in user key function by coding the following macro:

```
DATOFF INDMVCLK
```

INDXC0 and INDXC64 - Exclusive OR

INDXC0 input register values must be 31-bit addresses. INDXC64 input register values must be 64-bit addresses. Before issuing the macro, the user must load the registers as follows:

Register
Use

0 Used as a work register by the system

2 Real location of first operand and location for results of exclusive OR character operation

3 Length, in bytes, of operand pointed to by register 2. The length must be in bits 24-31 of register 3. Allows a maximum length of 256 bytes

4 Real location of the operand to be exclusive ORed with the operand pointed to by register 2.

14 Used as a work register by the system

15 Used as a work register by the system

1,5-13 Unused

The user invokes the 31-bit XC function by coding the following macro:

```
DATOFF INDXC0
```

INDCDS and INDCDS64 - Compare double and swap

INDCDS input register values must be 31-bit addresses. INDCDS64 input register values must be 64-bit addresses. Before issuing the macro, the user must load the registers as follows:

Register
Use

0 Used as a work register by the system

1 Unchanged

2,3 First 64 bit operand in even-odd pair of registers (target data)

4,5 Third 64 bit operand in even-odd pair of registers (source data)

6 Real address of second operand, a doubleword in storage (target address)

7-13 Unchanged

14 Used as a work register by the system

15 Used as a work register by the system

The user invokes the 31-bit CDS function by coding the following macro:

```
DATOFF INDCDS
```

Writing user DAT-OFF routines

As shown in 1 on page 384, there are eight DAT-OFF indexes that users can define. These indexes are INDUSR1, INDUSR2, INDUSR3, INDUSR4, INDUSR641, INDUSR642, INDUSR643, and INDUSR644. The entry points corresponding to these indexes are IEAVEUR1, IEAVEUR2, IEAVEUR3, IEAVEUR4, IEAV64U1, IEAV64U2, IEAV64U3, and IEAV64U4, respectively.

User written DAT-OFF routines are restricted as follows:

- The user of the DATOFF macro must be in key 0, supervisor state, and executing with DAT turned on.
- DAT-OFF routines INDUSR1, INDUSR2, INDUSR3, and INDUSR4 must have the attributes AMODE=31 and RMODE=ANY.
- DAT-OFF routines INDUSR641, INDUSR642, INDUSR643, and INDUSR644 must have the attributes AMODE=64 and RMODE=ANY.
- The DAT-OFF routine must preserve register 0 because register 0 contains the return address of the module that issued the DATOFF macro.
- The DAT-OFF routine must use branch instructions to link to other DAT-OFF routines.
- The DAT-OFF function must use BSM 0,14 to return from INDUSR1, INDUSR2, INDUSR3, and INDUSR4.
- The DAT-OFF function must return via BR 14 from INDUSR641, INDUSR642, INDUSR643, and INDUSR644.

Placing user DAT-OFF routines in the DAT-OFF nucleus

To add 31-bit DAT-off code to the DAT-off nucleus, and execute the code, follow these steps:

1. Create a separate module containing the code that runs with DAT-off, as follows:
   - Use entry point IEAVEURn, where n is a number from 1 to 4. MVS reserves four entry points in the DAT-off nucleus for AMODE 31 user code.
   - Give the module AMODE 31 and RMODE ANY attributes.
   - Make sure the DAT-off code does not alter register 0; it contains the return address to the routine that issues the DATOFF macro.
   - Use BSM 0,14 as the return instruction.
2. Linkedit your DAT-off module (IEAVEURn) into SYS1.NUCLEUS. The member names are IEAVEDAT and IEAV2DAT; input to the linkage editor must include an ORDER and ENTRY control statement for entry point IEAVEDAT in both modules.
3. Within a DAT-on routine, code a DATOFF macro to invoke the module created in step 1:

   ```
   DATOFF  INDEX=INDUSRn
   ```

   The suffix of the index (n) is the same as the suffix of the DAT-off module‘s entry point, IEAVEURn. See *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* for details on coding the DATOFF macro.

Note: You will need to re-linkedit your IEAVEURn module(s) into the DAT-off nucleus if you re-install the MVS base control program.

To add 64-bit DAT-off code to the DAT-off nucleus, and execute the code, follow these steps:

1. Create a separate module containing the code that runs with DAT-off, as follows:
   - Use entry point IEAV64Un, where n is a number from 1 to 4. MVS reserves four entry points in the DAT-off nucleus for AMODE 64 user code. Give the module AMODE 64 and RMODE ANY attributes.
   - Give the module AMODE 31 and RMODE ANY attributes.
   - Make sure the DAT-off code does not alter register 0; it contains the return address to the routine that issues the DATOFF macro.
   - Use BR 14 as the return instruction.
2. Linkedit your DAT-off module (IEAVEURn) into SYS1.NUCLEUS. The member name is IEAV2DAT; input to the linkage editor must include an ORDER and ENTRY control statement for entry point IEAVEDAT.
3. Within a DAT-on routine, code a DATOFF macro to invoke the module created in step 1:

   ```
   DATOFF INDEX=INDUSR64n
   ```

   The suffix of the index (n) is the same as the suffix of the DAT-off module‘s entry point, IEAV64Un. See *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* for details on coding the DATOFF macro.

Note: You will need to re-linkedit your IEAV64Un module(s) into the DAT-off nucleus if you re-install the MVS base control program.

Obtaining information about CSECTs in the DAT-ON nucleus (NUCLKUP)

IPL processing places the CSECTs located in the DAT-ON nucleus in virtual storage and creates a map of them. The real addresses do not equal the virtual addresses and the real addresses are not necessarily contiguous. IPL processing loads the CSECTs into storage according to residency mode and according to whether they are read only or read/write. If the CSECT is assembled with RMODE=ANY, it is placed in the extended nucleus. Figure 55 on page 324 shows the virtual storage map of the DAT-ON nucleus.

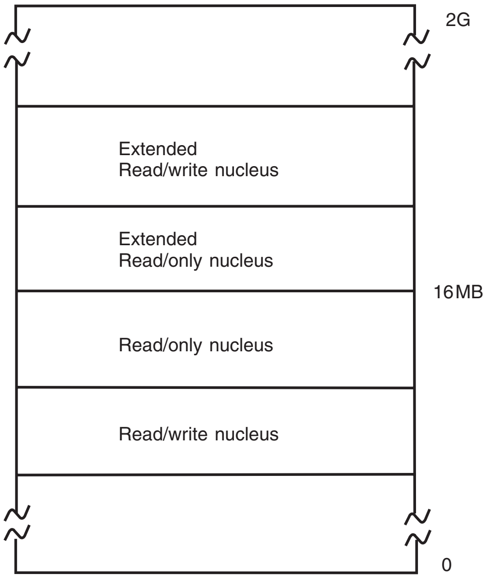


Figure 55. Virtual Storage Map of DAT-ON Nucleus

The nucleus map look up service provides users with information about these CSECTs. Through the use of the NUCLKUP macro, users can perform two functions:

- Retrieve the address and addressing mode of a nucleus CSECT, given the name of the CSECT
- Retrieve the name and entry point address of a nucleus CSECT, given an address within the CSECT.

Customizing the nucleus region

You can use the NUCLSTxx member of SYS1.PARMLIB or the NMLDEF macro to customize the nucleus region. Both are interfaces to the generalized nucleus loader, which:

- Enables you to maintain, as separate members in SYS1.NUCLEUS, modules to be loaded into the DAT-ON nucleus region
- Eliminates the need to link-edit, into an IEANUC0x member of SYS1.NUCLEUS, new nucleus-resident code such as installation-written SVCs.

You can design each NUCLSTxx parmlib member or nucleus module list (NML) to contain a list of the SYS1.NUCLEUS members that are part of the same product or user modification. The generalized nucleus loader processes all NMLs found in SYS1.NUCLEUS, including those used by IBM products and vendor products.

Note:

1. All multiple CSECT load modules, such as IEANUC0x, to be loaded into the nucleus region, must be link-edited with the scatter (SCTR) attribute. If this is not done, the system enters a non-restartable wait state at system initialization.
2. Neither the NUCLSTxx parmlib member nor the NMLDEF macro can be used to change the contents of IEANUC0x. They can be used only to specify which additional members of SYS1.NUCLEUS are to be loaded into the nucleus region along with IEANUC0x. See “Removing existing routines from IEANUC0x” on page 325 for information on deleting routines from IEANUC0x.

The NUCLSTxx parmlib member is easier to use and offers more flexibility than the NMLDEF macro. See *z/OS MVS Initialization and Tuning Reference* for more information about NUCLSTxx.

The NMLDEF macro

Use the NMLDEF macro to generate an NML. The macro generates a CSECT statement, the list of module names, and an END statement. The generated NML is a load module that resides in SYS1.NUCLEUS.

When using NMLDEF, ensure that:

- All SYS1.NUCLEUS members listed in an NML are in SYS1.NUCLEUS. If the system cannot find the specified SYS1.NUCLEUS member, it enters wait state X'055', with reason code X'2'.
- Each SYS1.NUCLEUS member is listed in only one NML. If the system finds a duplicate NML entry, it enters wait state X'025'.

See *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU* for details on coding the NMLDEF macro.

Removing existing routines from IEANUC0x

Nucleus-resident routines can be removed from an IEANUC0x member of SYS1.NUCLEUS by running a link-edit job to replace the existing version of IEANUC0x with one that does not contain the routine to be deleted.

Note:

1. If the existing nucleus-resident routine is known to SMP/E, use the SMP/E UCLIN statement to remove the module entry.
2. All multiple CSECT load modules, such as IEANUC0x, to be loaded into the nucleus region, must be link-edited with the scatter (SCTR) attribute. If this is not done, the system enters a non-restartable wait state at system initialization.

Chapter 18. Providing recovery

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

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

Part of recovery is also the "cleaning up" of any resources your program might have acquired. By "clean up" of resources, we mean that programs might need to release storage that was obtained, release ENQs, close data sets, and so on. If your program encounters an error before it has the opportunity to clean up resources, either your recovery routine, or another type of user-written routine called a **resource manager**, can do the clean up. You can also use a resource manager if your program ends normally without encountering any errors.

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

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

This information also discusses why you might want to provide resource managers to clean up resources acquired by your program, and how you go about doing that. You can provide resource managers whether or not you provide recovery routines. See “Using resource managers” on page 402 for further information.

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

To find out about:	Consult the following topic:
General recovery concepts, including: • Why you would want to provide recovery. • What software errors result in your recovery getting control. • What we mean when we say a program **abnormally ends**. • The different states for a recovery routine. • The different types of routines in a recovery environment, and how to choose, define, and activate the right recovery routine. • The basic options available to a recovery routine. • How routines in a recovery environment interact.	"Understanding general recovery concepts."
How to write a recovery routine, including: • What recovery routines do. • How recovery routines communicate with other routines and with the system. • Special considerations when writing different types of recovery routines.	"Writing recovery routines" on page 339.
The recovery environment, including: • Register contents at various times during recovery processing. • Other environmental factors such as program authorization, dispatchable unit mode, ASC mode, and so on.	"Understanding the recovery environment" on page 365.
Coding the various routines in a typical recovery environment.	"Understanding recovery through a coded example" on page 389.
Advanced recovery topics, including: • Providing recovery with minimal processor overhead (for types 2, 3, and 4 SVCs only). • Intentionally invoking RTM. • Providing multiple recovery routines. • Providing recovery for recovery routines. • Providing recovery for multitasking programs • Using resource managers.	"Understanding advanced recovery topics" on page 391.

Understanding general recovery concepts

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

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

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

Deciding whether to provide recovery

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

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

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

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

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

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

  Depending on the nature of the error, your recovery routine might successfully correct the error and allow your program to continue processing normally. Maintaining maximum availability is one of the major objectives of providing recovery.
- **Is a way to protect both system and application resources**.

  In general, recovery routines should clean up any resources your program is holding that might be requested by another program, or another user of your program. The purpose of clean up is to:
  - Allow your program to run again successfully without requiring a re-IPL
  - Allow the system to continue to run other work (consider especially other work related to the failing program).

  System locks, ENQs, and latches are examples of important resources shared by other programs. A program should provide for the release of these resources if an error occurs so that other programs can access them. Releasing resources is especially important if your program is a service routine. A service routine must release resources before returning to its caller, so the caller does not end up holding resources that it did not request.

  Note: Locks, ENQs, and latches are all used for serialization. See Chapter 4, “Serialization,” on page 25 for more information about serialization.

Another resource a program should release is any virtual storage it obtained, so that the storage becomes available to other programs. Note that the most important storage to release is common storage.

Recovery routines should also ensure the integrity of any data being accessed. Consider the case of a database application that is responsible for protecting its database resources. The application must ensure the integrity and consistency of the data in the event an error occurs. Data changes that were made prior to the error might have to be backed out from the database.

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

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

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

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

  Recovery routines can do such things as save serviceability data, request recording of an error in the logrec data set, and request dumps. Each of these actions help you determine what went wrong in your program, and each is explained in greater detail later in this information. Note that the recovery routine must provide whatever serviceability data it wants the system to record.

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

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

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

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

Understanding errors in MVS

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

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

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

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

- Unanticipated program checks (except those resolved by SPIE or ESPIE routines; see Chapter 19, “Processing program interruptions (SPIE, ESPIE),” on page 413 and “Program Interruption Services” in *z/OS MVS Programming: Assembler Services Guide* for information about SPIE and ESPIE routines.)
- Machine checks (such as a storage error that occurs while your program is running)
- Various types of CANCEL (such as operator or time out)
- An error when issuing an MVS macro or callable service (for example, specifying parameters that are not valid)

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

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

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

Use the CALLRTM macro to request recovery processing on behalf of a particular task or address space. Your program might do this if, for example, the program is sharing resources with another task, and that task appears to have stopped processing while holding an exclusive ENQ on a resource. Your program might issue the CALLRTM macro to signal the system to pass control to a recovery routine so the resource can be freed.

Understanding recovery routine states

In this information, recovery routines can be in one of the following states:

- **Defined**

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

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

  A recovery routine is **in control** when it is running; an error has occurred and the system passed control to the recovery routine.
- **No longer in control**

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

- **Deactivated**

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

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

Understanding the various routines in a recovery environment

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

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

All of these routines are user-written routines.

Mainline routine

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

Recovery routine

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

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

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

Retry routine

A retry routine is essentially an extension of the mainline routine. When an error occurs, the system passes control to your recovery routine, which can then request the system to pass control back to the mainline routine to resume processing. That portion of the mainline that gets control back is referred to as the retry routine. When the retry routine gets control, it is as if the mainline routine branched there after encountering the error; to the mainline routine, it appears as though the error never occurred.

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

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

Resource manager

A resource manager is a routine you can write to handle the clean up of resources owned by a task or address space. Resource managers get control only when all programs running under a particular task or address space have ended. Resource managers get control when a task or address space is ending either normally or abnormally.

Choosing the appropriate recovery routine

Once you have decided to provide recovery for your program, you need to decide what type of recovery to provide. Recovery routines fall into two broad categories: **ESTAE-type recovery routines** and **FRRs**. In turn, there are different types of ESTAE-type recovery routines. This information describes the different types of recovery routines, and for each type describes how you define it, activate it, deactivate it, and request that it be no longer defined. A summary of this information is in Table 39 on page 336.

When you provide one or more recovery routines for your program, you have the opportunity to identify a user parameter area for the system to pass from the mainline routine to the recovery routine. Creating such a parameter area with information for the recovery routine is a very important part of providing recovery. See “Setting up, passing, and accessing the parameter area” on page 348 for more information about what this parameter area should contain, and how to pass it.

Choosing an ESTAE-type recovery routine

Use ESTAE-type recovery routines to protect programs running under enabled, unlocked tasks (EUTs). ESTAE-type recovery routines can be in either problem state or supervisor state, and can have any PSW key.

Programs that are disabled, hold locks, or are in SRB mode cannot use ESTAE-type recovery routines. If your program obtains a lock after you activate an ESTAE-type recovery routine, and then encounters an error causing your recovery routine to get control, the system releases the lock. You could use an ESTAE-type recovery routine in this situation if your program can tolerate losing its lock. For example, the lock might be used only to protect a queue from change while it is being read.

Whatever is said about ESTAE-type recovery routines throughout this information applies to recovery routines defined in all of the following ways, unless stated otherwise:

- STAE, ESTAE, and ESTAEX macros

- ATTACH and ATTACHX macros with STAI and ESTAI parameters
- ETDEF macro (ARR parameter, together with the ETCRE macro) or IEAARR macro
- FESTAE macro.

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

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

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

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

  – The recovery routine fails to or chooses not to retry (explained under "Understanding recovery routine options" on page 336).

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

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

  IBM recommends you always use IEAARR or ESTAEX unless your program and your recovery routine are in 24-bit addressing mode, or your program requires a branch entry. In these cases, you should use ESTAE. ESTAE and ESTAEX provide the same function, except that ESTAEX can be issued in AR ASC mode and in cross memory mode.

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

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

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

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

  IBM recommends you always use the ESTAI, rather than the STAI, parameter on ATTACHX, rather than ATTACH. ATTACH and ATTACHX provide the same function, except that ATTACHX can be issued in AR ASC mode.

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

- **ETDEF macro (ARR parameter, together with the ETCRE macro) or IEAARR macro**

  To provide recovery for a stacking PC routine, the program defining the stacking PC routine can use the ARR parameter on the ETDEF macro to identify an associated recovery routine (ARR). Using the ETDEF macro with the ARR parameter, together with the ETCRE macro, defines the ARR to the system. You can also cause the system to establish an ARR to cover a target routine by issuing the IEAARR macro. When you issue the PC instruction to give control to the stacking PC routine, the ARR is activated. The ARR remains activated, and cannot be deactivated, while the stacking PC routine is running. The ARR receives control if the stacking PC routine encounters an error. The ARR is deactivated when the stacking PC routine issues the PR instruction to return control to its caller. When you issue the ETDES macro, the ARR is no longer defined.

Note: Use an ARR only if the stacking PC routine is always entered in enabled, unlocked task mode. See “Using ARRs” on page 361 for more information about ARRs.

- **FESTAE macro**

  The recovery routine you define with the FESTAE macro is similar to an ESTAE routine, but is limited to types 2, 3, and 4 SVCs that require recovery with minimal processor overhead.

  You both define and activate the recovery routine by issuing the FESTAE macro with the EXITADR parameter. You deactivate the recovery routine and request that it be no longer defined by issuing FESTAE with the 0,WRKREG parameter.

  The remainder of this information refers to the recovery routines you define through the FESTAE macro as **FESTAE routines**. See “Providing recovery with minimal processor overhead (FESTAE macro)” on page 392 for more information about using FESTAE.

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

All **ESTAE-type recovery routines** handle the Floating Point Registers (FPRs) and Floating Point Control (FPC) register the same as for FRRs (see below), except that the DXC field in the Floating Point Control register could have been altered by an asynchronous exit. Also, in general ESTAE-type routines should not depend on the FPRs and FPC register containing the time-of-error values.

Choosing an FRR

A supervisor state and PSW key 0-7 program must choose an FRR, as opposed to an ESTAE-type recovery routine, if the program is disabled, locked, or in SRB mode. To both define and activate the FRR, the program issues the SETFRR macro.

Note: Supervisor state and PSW key 0-7 programs can be protected by FRRs, but the program must be in supervisor state and PSW key 0 when it issues the SETFRR macro.

A supervisor state and PSW key 0-7 program can also choose an FRR if it is enabled, unlocked, and in task mode, by specifying EUT=YES on SETFRR. An FRR defined in this way is called an EUT FRR. A program might choose an EUT FRR if the circumstances under which the program will get control are unknown. If you do not know if the program will get control in task or SRB mode, whether it will be disabled, or whether it will be holding a lock, you can use an EUT FRR.

Once a program activates an EUT FRR, the program cannot issue any SVCs, and cannot handle asynchronous interrupts (the system does not dispatch any new asynchronous exits on that task until all FRRs for the task have been deactivated.)

An FRR is activated until one of the following events occurs:

- You deactivate it and request that it be no longer defined (issue SETFRR D)

- The FRR fails to or chooses not to retry (explained under "Understanding recovery routine options")
- The unit of work (either task or SRB) under which the caller of SETFRR is running terminates.

From the time an FRR is activated until the time it is deactivated, **at least one of the following must be true**:

- The unit of work holds a lock
- The unit of work is running disabled
- The unit of work is an SRB
- The FRR is an EUT FRR.

If at any time none of the above conditions are met, then the results are unpredictable, and your program might lose its FRR recovery. Note also that any FRR entered disabled **must remain disabled**.

Floating point implications

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

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

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

Summary of recovery routine states

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

Table 39. Summary of Recovery Routine States

Recovery routine	Defined	Activated	Deactivated	No longer defined
ESTAE	ESTAE CT	ESTAE CT	ESTAE 0	ESTAE 0
ESTAEX	ESTAEX CT	ESTAEX CT	ESTAEX 0	ESTAEX 0
ESTAI	ATTACHX ESTAI	Attached task gets control	Attached task ends	Attached task ends
ARR	ETDEF ARR and ETCRE	PC instruction	PR instruction	ETDES
ARR	IEAARR	system-issued PC instruction	system PR instruction	system PR instruction
FESTAE	FESTAE	FESTAE	FESTAE 0	FESTAE 0
FRR	SETFRR A	SETFRR A	SETFRR D	SETFRR D

Understanding recovery routine options

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

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

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

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

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

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

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

See “Providing multiple recovery routines” on page 395 for more information about having multiple recovery routines.

If no other recovery routines are activated, or all recovery routines percolate, the system proceeds to abnormally end your program.

Understanding how routines in a recovery environment interact

Figure 56 on page 338 is a very simplified illustration of how routines in a recovery environment interact. In this figure, only one recovery routine exists, and it is an ESTAE-type recovery routine. The following sequence of events might occur:

1. The mainline routine encounters an error.
2. The system gets control.
3. The system looks for recovery routines and finds an ESTAE-type recovery routine called ESTAEX.
4. The ESTAEX routine either retries or percolates.
   a. If the ESTAEX routine retries, it returns control to a retry point in the mainline routine. The mainline routine continues processing.
   b. If the ESTAEX routine percolates, the system gets control and abnormally ends the mainline routine; then the system gives control to any resource managers that are present.

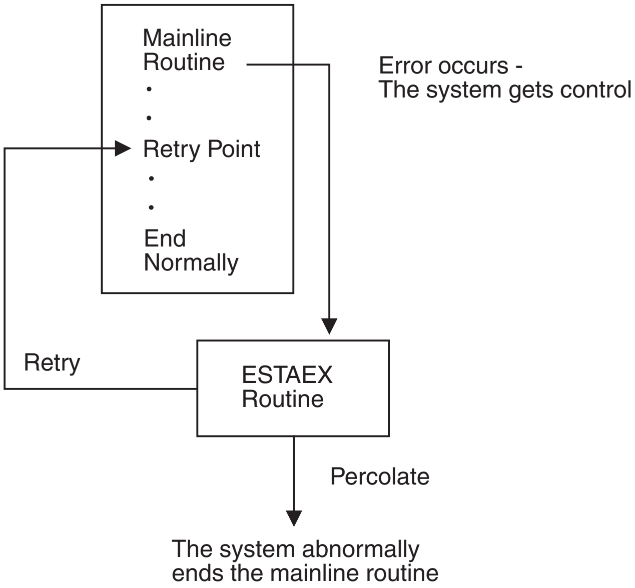


Figure 56. Mainline Routine with One Recovery Routine

Figure 57 on page 339 shows a more complex situation. Several recovery routines exist, and each one that is entered has the opportunity to retry or to percolate. The following sequence of events might occur if all recovery routines percolate:

1. The mainline routine encounters an error.
2. The system looks for recovery routines and finds that the latest FRR created is FRR(3).
3. The system gives control to FRR(3) first.
4. FRR(3) percolates to FRR(2), which percolates to FRR(1).
5. FRR(1) also percolates, so the system looks for ESTAE-type recovery routines, and finds that ESTAEX(4) was the last one created.
6. The system gives control to ESTAEX(4) first.
7. ESTAEX(4) percolates to ARR(3), which percolates to ESTAE(2), which percolates to ESTAI(1).
8. ESTAI(1) also percolates, and no other recovery routines are activated, so the system abnormally ends the mainline routine, and passes control to any resource managers that are present.

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

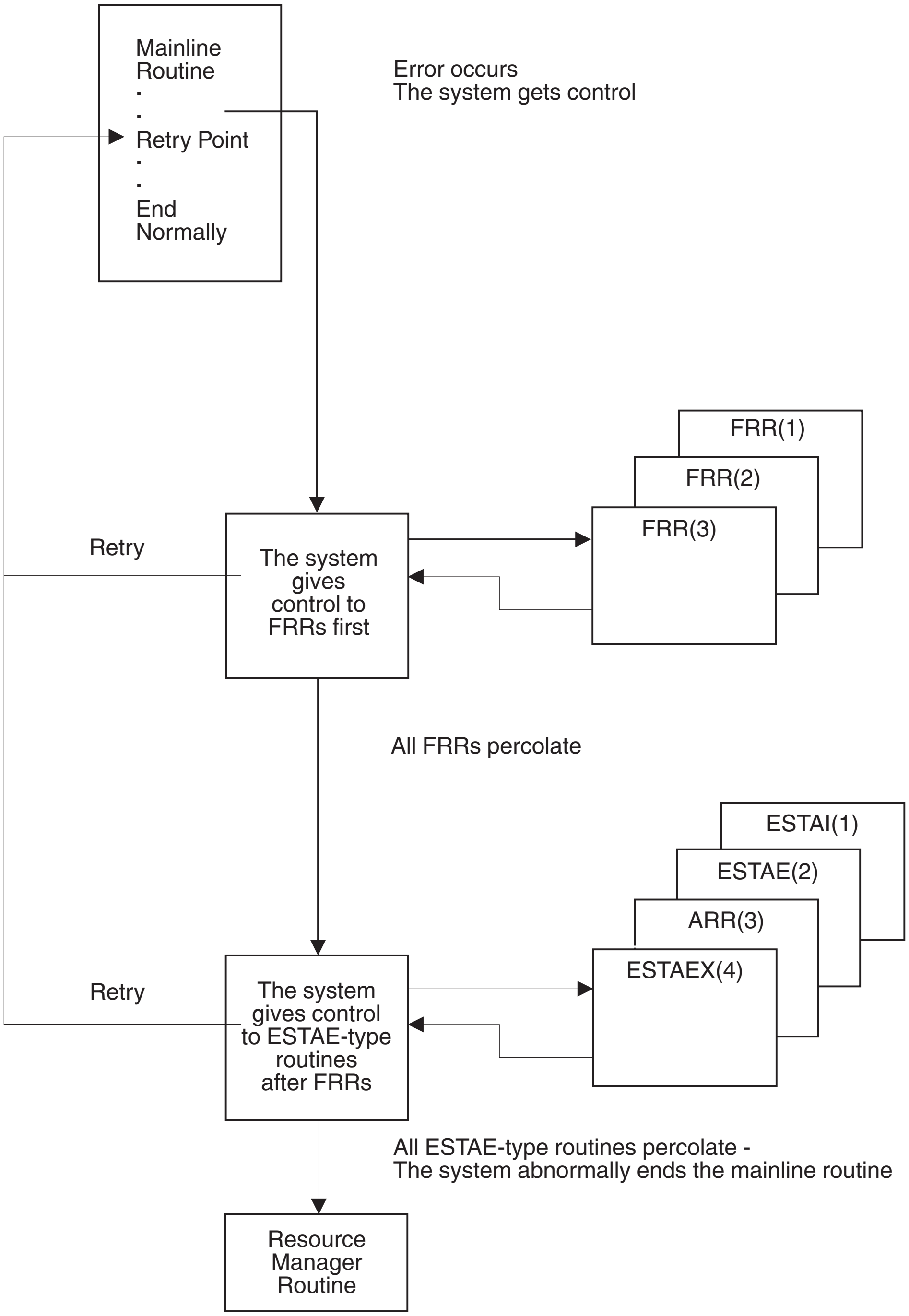


Figure 57. Mainline Routine with Several Recovery Routines

Writing recovery routines

This information has discussed general recovery concepts, including how to decide what type of recovery you need, and how to provide that recovery. To write the recovery routines that you provide, you must understand all of the following:

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

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

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

  The means of communication available to a recovery routine are:
  - A user parameter area, built by the mainline routine and passed to the recovery routine.
  - A data area called the system diagnostic work area (SDWA), which is provided by the system. The recovery routine communicates with the system, with other recovery routines, and with the retry routine through the SDWA. The recovery routine uses the SETRP macro to update information in the SDWA.
  - Registers, when no SDWA is provided.
- **The special considerations you must make when writing an ESTAE-type recovery routine as opposed to an FRR.**

  One important consideration is the presence of an SDWA. When an FRR gets control, the system always provides an SDWA; that is not true for an ESTAE-type recovery routine. The case where an SDWA is not provided is rare; nevertheless, when you design an ESTAE-type recovery routine, you must allow for the possibility of not receiving an SDWA.

  Remember also that programs running disabled, holding a lock, or in SRB mode must use an FRR rather than an ESTAE-type recovery routine.

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

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

Understanding what recovery routines do

The following is a list of some of the things a recovery routine should do if the recovery is to be effective.

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

- Preserve the return address to the system.
- Check for the presence of an SDWA (for ESTAE-type recovery routines only).
- Establish addressability to the parameter area passed by the mainline routine. How you do that depends on whether an SDWA is present.
- Check the contents of important fields in the SDWA.
  - Determine the location of the parameter area.
  - Determine why the routine was entered.
  - Determine if this is the first recovery routine to get control.
- Check the contents of the parameter area passed by the mainline.
  - Determine if this is a repeated error (to avoid recursion).
  - Determine when and where the error occurred.

- Provide information to help determine the cause of the error:
  - Save serviceability data in the SDWA.
  - Request recording in the logrec data set.
  - Request a dump of storage.
- Try to correct or minimize the effects of the error.
- Determine whether the recovery routine can retry, decide whether to retry or percolate, and take the appropriate actions (such as cleaning up resources).

Saving the return address to the system

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

Checking for the SDWA

Even though this applies only to ESTAE-type recovery routines, it is important enough to emphasize here in the general information.

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

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

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

FRRs do not have to check for the presence of an SDWA. When FRRs get control, the system always provides an SDWA.

Establishing addressability to the parameter area

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

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

Refer to “Setting up, passing, and accessing the parameter area” on page 348 for further detail on accessing the parameter area.

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

- A dynamic storage area

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

Checking important fields in the SDWA

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

- **Why the routine was entered.**

  The routine can check the SDWACMPC field, which contains the completion code that existed when the system gave control to the routine, and the SDWACRC field, which contains the reason code associated with the completion code. SDWACRC contains a reason code only if the SDWARCF bit is on.
- **The location of the parameter area that was passed by the mainline.**

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

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

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

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

  The SDWAPERC bit is significant only between similar types of recovery routines. An ESTAE-type recovery routine can tell only if another ESTAE-type recovery routine has percolated; an FRR can tell only if another FRR has percolated.

See “Important fields in the SDWA” on page 353 for a list of some of the fields in the SDWA, and an explanation of their contents.

Checking the contents of the parameter area

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

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

The recovery routine can tell when and where the error occurred through “footprints,” a technique explained under “Deciding what to include in the parameter area” on page 348. Footprints can help the recovery routine to avoid getting into a loop in which the routine requests a retry, and the same error occurs again (recursion). For example, if the recovery routine supplies a bad retry address to the system, and the processing of the first instruction at the given address causes a program check, the first recovery routine to get control is the one that just requested the retry. If the recovery routine requests another retry at the same address, the loop is created.

Note: For an FRR, the system initializes the parameter area to zeros. If an FRR receives control when the parameter area still contains zeros, the error probably occurred before the mainline code initialized the parameter area.

See “Setting up, passing, and accessing the parameter area” on page 348 for more information about what the parameter area can contain, and the techniques you can use to provide the most useful information to the recovery routine.

Saving serviceability data

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

- By issuing the SETRP macro with the RECPARM parameter. Use the RECPARM parameter to supply the load module name, the active CSECT name, and the recovery routine CSECT name. See “Using the SETRP macro to update the SDWA” on page 352 for more information about using SETRP.
- By issuing the VRADATA macro to update the SDWA variable recording area. See the VRADATA macro in *z/OS MVS Programming: Assembler Services Reference IAR-XCT* for more information.
- By directly manipulating other fields in the SDWA. Important fields to fill in are SDWACID, SDWASC, SDWAMLVL, and SDWARRL. See “Important fields in the SDWA” on page 353 for a description of each of these fields.

Part of saving serviceability data includes providing information for dump analysis and elimination (DAE). DAE depends on information that users provide in recovery routines to construct symptom strings needed to describe software failures. DAE uses these symptom strings to analyze dumps and suppress duplicate dumps as requested. You should provide information for DAE prior to requesting a dump of storage. See “Suppressing SVC dumps that duplicate previous SVC dumps” on page 424 for more information about DAE and dump suppression.

Recording in the Logrec data set

The system records the entire SDWA (including the fixed length base, the variable length recording area, and the recordable extensions) in the logrec data set under the following circumstances, depending on the type of recovery routine:

- **FRRs**

  The system automatically records the SDWA in the logrec data set for the first FRR, unless the FRR specifies RECORD=NO on the SETRP macro. Subsequent FRRs that require recording in the logrec data set must specify RECORD=YES on SETRP.
- **ESTAE-type recovery routines**

  The system does not automatically record the SDWA in the logrec data set for any ESTAE-type recovery routines. The mainline routine can request recording through the RECORD parameters on the ESTAE, ESTAEX, and FESTAE macros. The recovery routine can request recording through the RECORD parameter on the SETRP macro.

  The recovery routine can also override whatever the mainline routine requests. If the recovery routine specifies RECORD=YES (or RECORD=NO) on SETRP, the system will (or will not) record the SDWA in the logrec data set, regardless of what the mainline routine specifies on ESTAE, ESTAEX, or FESTAE. The recovery routine can honor what the mainline routine requests by specifying RECORD=IGNORE on SETRP.

Note: The routine that defines an ESTAI routine or an ARR cannot request recording. ESTAI routines and ARRs can request recording only through the SETRP macro.

Requesting a dump

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

"Dumping Virtual Storage (ABEND, SNAPX, and SNAP Macros" in *z/OS MVS Programming: Assembler Services Guide* contains information about ABEND dumps. Chapter 20, "Dumping virtual storage (SDUMPX, SDUMP, and IEATDUMP macros)," on page 417 and Chapter 19, "Processing program interruptions (SPIE, ESPIE)," on page 413 contain information about SVC dumps.

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

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

Correcting or minimizing the error

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

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

  For example, one of the data areas might contain an address that is not valid. Or, the mainline routine might have set up the data areas with some sort of validating information that could be checked, and possibly corrected. Certain data areas might have to be deleted from the queue, or the entire queue might have to be deleted and rebuilt.
- The mainline routine might be running under a task that is communicating with another task when an error occurs. The recovery routine might then take the action of alerting the other task that a problem exists, so the other task does not wait for any further communication.

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

Deciding to retry or percolate

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

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

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

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

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

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

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

- Eliminate or minimize the cause of the error with complete or partial repair, as explained above under “Correcting or minimizing the error” on page 344.
- Ensure that the retry routine's environment is restored. For example, restore registers and re-establish addressability to mainline resources. See “Register contents on entry to a retry routine” on page 369 for details about how a recovery routine can control the register contents on entry to the retry routine.
- Know the condition of resources being held by the mainline. For example, the recovery routine might have to acquire locks to repair data structures, back out changes to data sets, and so on.
- For FRRs, decide what to do about locks that were acquired by the mainline and are held on entry to the FRR. In general, the FRR's responsibility is to make sure that all locks are freed, either by itself, or by the system. See “Special considerations for FRRs” on page 363 for details about freeing locks held on entry to an FRR. ESTAE-type recovery routines are not entered holding any locks.
- If the recovery routine obtains any locks after it gets control, it should free them before returning control to the system. If an FRR returns to the system holding

locks the FRR obtained, the system frees them. An ESTAE-type recovery routine **must** free any locks it obtains before returning control to the system or an error occurs.

- Indicate to the system that a retry is to be attempted. If an SDWA is present, the recovery routine issues the SETRP macro with the RC=4 parameter to indicate retry, and the RETADDR parameter to specify the address of the retry routine. You can specify RC=4 even when the SDWACLUP bit is on, indicating that retry is not allowed. If you do so, however, the system ignores the retry request.

  If no SDWA is present, the recovery routine has to set a return code of 4 in GPR 15, and place the address of the retry routine in GPR 0.
- For an ESTAE-type recovery routine, decide whether to pass the SDWA to the retry routine, and so indicate on the SETRP macro with the FRESDWA parameter.

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

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

- The retry routine runs under the same unit of work that activated the recovery routine. See “Special considerations for ESTAE-type recovery routines” on page 357 for further details related to ESTAE-type recovery routines.
- The retry routine might or might not have access to the SDWA, and the recovery routine might or might not have directed that register contents be restored for the retry routine.

  For ESTAE-type recovery routines that specify FRESDWA=YES on SETRP, and for all FRRs, the system frees the SDWA before entering the retry routine.

  For ESTAE-type recovery routines that specify RETREGS=YES, and for all FRRs, the system restores the registers from the SDWA.

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

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

  When the retry routine is to free the SDWA, note that an ESTAE-type recovery routine activated under PSW key 0 - 7 receives an SDWA in key 0 storage. If the retry routine is running under a key other than key 0, it must do one of the following:

  - Issue the MODESET macro to switch into PSW key 0 before freeing the SDWA.
  - Free the SDWA using the STORAGE macro with the KEY parameter to specify a storage key of 0.

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

Table 40. Contents of GPR 0 on Entry to a Retry Routine

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

For complete details about register contents see “Understanding the recovery environment” on page 365.

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

Note:

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

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

- Release resources that were acquired by the mainline, such as:
  - Storage that should be explicitly freed (for example, CSA)
  - ENQs
  - Locks (the same rules regarding locks apply to recovery routines that percolate as those described under “Recovery routines that retry” on page 345.)
- Repair the cause of the error, if possible.
- Indicate the percolate option to the system. If an SDWA is present, the recovery routine issues the SETRP macro with the RC=0 parameter to indicate percolation. If no SDWA is present, the recovery routine has to set a return code of 0 in register 15.

Note:

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

Understanding the means of communication

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

- **Parameter area**

  The parameter area is set up by the mainline routine and passed to the recovery routine. See "Setting up, passing, and accessing the parameter area."
- **SDWA**

  The SDWA provides information to the recovery routine, and the recovery routine can communicate with the system, and with subsequent recovery routines, by placing information into the SDWA. See "Using the SDWA" on page 352.
- **Registers**

  When a recovery routine gets control, GPR 0 indicates whether an SDWA is available. When an SDWA is not available, the recovery routine can communicate its recovery options to the system only through registers. Aside from this circumstance, the recovery routine **cannot** use registers to communicate with the system; the routine must use the SDWA. Also, the mainline routine should not place information in registers and expect that information to be in the registers when the recovery routine gets control. Complete details about registers are in "Understanding the recovery environment" on page 365.

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

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

Setting up, passing, and accessing the parameter area

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

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

Deciding what to include in the parameter area: Your mainline routine can put whatever information it wants in the parameter area. Remember that the object is to provide the recovery routine with as much useful information as possible so the recovery routine can be effective. Here are some suggestions for important information to place in the parameter area:

- The base registers for the mainline. The recovery routine must be able to establish addressability to whatever resources the mainline is holding.
- The addresses of all dynamically acquired storage.

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

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

- **ESTAE, ESTAEX, and FESTAE routines**

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

  Note: For a recovery routine defined by a FESTAE macro, there is also a 24-byte parameter area available in the SVRB, in field RBFEPARM. See "Providing recovery with minimal processor overhead (FESTAE macro)" on page 392 for more information about RBFEPARM.
- **ESTAI routines**

  Use the ESTAI parameter on the ATTACHX macro to specify both the address of the recovery routine to get control, and the address of the parameter area you have constructed.
- **ARRs**
  - If the ARR will not be given control in AMODE 64, the stacking PC routine for which the ARR was defined can use the 8-byte modifiable area of the linkage stack to provide the address and ALET of the parameter area to be passed to the recovery routine.
  - If the ARR will be given control in AMODE 64, the stacking PC routine for which the ARR was defined can use the 8-byte modifiable area of the linkage stack to provide the 64-bit address of the parameter area to be passed to the recovery routine.

The stacking PC routine places the address and ALET into an even/odd GPR pair and then uses the MSTA instruction to move the information into the linkage stack.

- **IEAARR routines**

  Use the ARRPARAMPTR parameter on the IEAARR macro to specify the 31-bit address of the parameter area you constructed, or use the ARRPARAMPTR64 parameter to specify the 64-bit address of the parameter area you constructed.
- **FRRs**

  Use the PARMAD parameter on the SETFRR macro to specify a 4-byte field or GPR into which the system places the address of a 24-byte parameter area, initialized to zeros. You use this system-provided parameter area to place the parameter information to be passed to the FRR.

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

- **SDWA is present**
  - **ESTAE and FESTAE macros**

    SDWAPARM and GPR 2 contain the address of the parameter area you specified on the PARAM parameter on ESTAE or FESTAE. For FESTAE, if you did not specify PARAM, either SDWAPARM or GPR 2 contains the address of the 24-byte parameter area in the SVRB (RBFEPARM).
  - **ESTAEX macro**

    SDWAPARM contains the address of an 8-byte field, which contains the address and ALET of the parameter area you specified on the PARAM parameter on ESTAEX, and GPR 2 contains the address of the parameter area you specified on the PARAM parameter on ESTAEX. AR 2 contains the ALET qualifying the address in GPR 2.
  - **ESTAEX macro issued in AMODE 64**

    SDWAPARM contains the address of an 8-byte area, which contains the address of the parameter area you specified on the PARAM parameter of ESTAEX. GPR 2 contains the 64-bit address of the parameter area.
  - **ATTACHX macro with ESTAI parameter**

    SDWAPARM and GPR 2 contain the address of the parameter area you specified on the ESTAI parameter on ATTACHX. When ATTACHX is issued in AMODE 64 the parameter list address is still treated as a 31-bit address. The parameter area specified on ATTACHX is always assumed to be in the primary address space, so for AR-mode callers, the ALET is always zero.
  - **ARRs not in AMODE 64**

    SDWAPARM contains the address of a copy of the 8-byte modifiable area of the linkage stack, and GPR 2 contains a copy of the first 4 bytes from the 8-byte modifiable area of the linkage stack; AR 2 contains a copy of the second 4 bytes from the 8-byte modifiable area of the linkage stack. The stacking PC routine covered by the ARR should have set the contents of this modifiable area to contain the address and ALET of the parameter area by using the MSTA instruction.
  - **ARRs in AMODE 64**

    SDWAPARM contains the address of a copy of the 8-byte modifiable area of the linkage stack, and 64-bit GPR 2 contains a copy of that 8-byte modifiable area; AR 2 contains 0. The stacking PC routine covered by the ARR should have set the contents of this modifiable area to contain the 64-bit address of the parameter area by using the MSTA instruction.

  - **IEAARR macro**

    SDWAPARM contains the address of an 8-byte area. The first word of this area contains the address of the parameter area you specified on the ARRPARAMPTR parameter of IEAARR and the second word does not contain interface information. GPR 2 contains the address of the parameter area.

  - **IEAARR macro issued in AMODE 64**

    SDWAPARM contains the address of an 8-byte area, which contains the address of the parameter area you specified on the ARRPARAMPTR64 parameter of IEAARR. GPR 2 contains the 64-bit address of the parameter area.

  - **FRRs**

    SDWAPARM and GPR 2 contain the address of the 24-byte parameter area returned by the SETFRR macro when you specified PARMAD.

- **SDWA is not present**
  - **ESTAE and FESTAE macros**

    GPR 2 contains the address of the parameter area you specified on the PARAM parameter on ESTAE or FESTAE. For FESTAE, if you did not specify PARAM, GPR 2 contains the address of the 24-byte parameter area in the SVRB (RBFEPARM).

  - **ESTAEX macro**

    GPR 2 contains the address of the parameter area you specified on the PARAM parameter on ESTAEX. AR 2 contains the ALET qualifying the address in GPR 2.

  - **ESTAEX macro issued in AMODE 64**

    GPR2 contains the 64-bit address of the parameter area you specified on the PARAM parameter of ESTAEX.

  - **ATTACHX macro with ESTAI parameter**

    SDWAPARM and GPR 2 contain the address of the parameter area you specified on the ESTAI parameter on ATTACHX. When ATTACHX is issued in AMODE 64 the parameter list address is still treated as a 31-bit address. The parameter area specified on ATTACHX is always assumed to be in the primary address space, so for AR-mode callers, the ALET is always zero.

  - **ARRs not in AMODE 64**

    GPR 2 contains a copy of the first 4 bytes from the 8-byte modifiable area of the linkage stack; AR 2 contains a copy of the second 4 bytes from the 8-byte modifiable area of the linkage stack. The stacking PC routine covered by the ARR should have set the contents of this modifiable area to contain the address and ALET of the parameter area by using the MSTA instruction.

  - **ARRs in AMODE 64**

    64-bit GPR 2 contains a copy of the 8-byte modifiable area of the linkage stack; AR 2 contains 0. The stacking PC routine covered by the ARR should have set the contents of this modifiable area to contain the 64-bit address of the parameter area by using the MSTA instruction.

  - **IEAARR macro**

    GPR 2 contains the address of the parameter area you specified on the ARRPARAMPTR parameter of IEAARR.

  - **IEAARR macro issued in AMODE 64**

GPR 2 contains the 64-bit address of the parameter area you specified on the ARRPARAMPTR64 parameter of IEAARR.

– **FRRs**

FRRs always get control with an SDWA.

Using the SDWA

The SDWA is both a means by which the recovery routine can provide information to the system and to subsequent recovery routines, and a provider of information to the recovery routine. To access and update the SDWA, the recovery routine must include the IHASDWA mapping macro as a DSECT. For complete information about the SDWA, see SDWA in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). **The SDWA is always in the primary address space.**

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

- By issuing the SETRP macro (See "Using the SETRP macro to update the SDWA.")
- By issuing the VRADATA macro (See the VRADATA macro in *z/OS MVS Programming: Assembler Services Reference IAR-XCT* and information about symptoms provided by a recovery routine in *z/OS MVS Programming: Assembler Services Guide*.)
- By directly updating specific fields (see "Important fields in the SDWA" on page 353).

Recording the SDWA in the logrec data set: A recovery routine can request the system to record the SDWA in the logrec data set by specifying the RECORD parameter on SETRP. See "Recording in the Logrec data set" on page 343 for more information.

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

- **Indicate retry or percolate**

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

  For ESTAI routines, you can also specify RC=16 to ask the system not to give control to any further ESTAI routines.
- **Specify register contents for the retry routine and free the SDWA**

  ESTAE-type recovery routines can use parameters on the SETRP macro to restore registers from the SDWA (RETREGS=YES), and to free the SDWA before control is given to the retry routine (FRESDWA=YES). See "Retry from an ESTAE-type recovery routine" on page 369 for information about using the RETREGS and FRESDWA parameters.

  For FRRs, RETREGS=YES or RETREGS=NO has no effect; the system always restores registers from the SDWA. FRRs cannot specify the FRESDWA parameter; the system always frees the SDWA before giving control to the retry routine.
- **Save serviceability data**

  Use the RECPARM parameter to supply the load module name, the active CSECT name, and the recovery routine CSECT name.
- **Change the completion and reason codes**

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

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

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

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

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

Symptom data required in the SDWA for dump suppression: If the installation is using DAE to suppress duplicate dumps, the recovery routine must provide the following minimum data to enable dump suppression. See *z/OS MVS Programming: Assembler Services Guide* for more information about dump suppression.

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

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

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

The fields described below that are in SDWARC1 are:

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

Field Name
 Use

SDWAPARM
For ESTAEX, if the ESTAEX was established by a routine running in AMODE 64, the 8-byte area contains the address of the parameter area you specified on the PARAM parameter of the ESTAEX macro. Otherwise, the

first four bytes of this 8-byte area contain the address of the parameter area and the next four bytes contain the ALET for the parameter area.

For routines defined by the IEAARR macro, if the IEAARR was issued in AMODE 64, the 8-byte area contains the 64-bit address of the parameter area specified on the ARRPARMPTR64 parameter of IEAARR. Otherwise, this field contains the address of an 8-byte area. The first word of this area contains the address of the parameter area you specified on the ARRPARAMPTR parameter of IEAARR and the second word does not contain interface information.

See "Setting up, passing, and accessing the parameter area" on page 348 for details on the parameter area passed by recovery routines.

SDWACMPC
This 3-byte field contains the completion code that existed when the system gave control to the recovery routine. The recovery routine can change the completion code by issuing the SETRP macro with the COMPCOD parameter. The system completion code appears in the first twelve bits, and the user completion code appears in the second twelve bits.

SDWARPIV
This bit tells the recovery routine that the registers and PSW at the time of error are not available. When this bit is on, the contents of SDWAGRSV, SDWAG64, SDWAARER, and SDWAEC1 are unpredictable.

SDWACRC
This 4-byte field contains the reason code associated with the completion code in SDWACMPC. The reason code is set through the REASON parameter of the CALLRTM or ABEND macro, and is valid only when bit SDWARCF is on. The recovery routine may change this reason code by specifying a new value for the REASON parameter of the SETRP macro.

Note: This reason code is **not** the same as the return code that programs may set in GPR 15 before they issue the CALLRTM or ABEND macro.

SDWARCF
If on, this bit indicates that SDWACRC contains a reason code.

SDWAGRSV
This field contains the contents of the general purpose registers (GPRs) 0-15 as they were at the time of the error.

SDWAARER
This field contains the contents of the access registers (ARs) 0-15 as they were at the time of the error.

SDWAEC1
This field contains the PSW that existed at the time of the error.

SDWAEC2
The contents of this field vary according to the type of recovery routine:

- For ESTAE-type recovery routines (except for ESTAI routines): If a program establishes an ESTAE routine, and subsequently performs a stacking operation while running under the same RB as when it established the ESTAE routine, SDWAEC2 contains the PSW from the linkage stack entry immediately following the entry that was current when the ESTAE routine was established. Otherwise, SDWAEC2 contains the current RBOPSW from the RB that activated the recovery routine, and the PSW is the one from the time of the last interruption of that RB that occurred while the RB was unlocked and enabled. Bit SDWAINTF in SDWAXFLG indicates whether the contents of SDWAEC2 are from the linkage stack (SDWAINTF is 1) or from an RB (SDWAINTF is 0).

- For an ESTAI routine, this field contains zero.
- For an FRR, the field contains the PSW used to give control to the FRR.

SDWASRSV

The contents of this field vary according to the type of recovery routine:

- For ESTAE-type recovery routines (except for ESTAI routines): If a program establishes an ESTAE routine, and subsequently performs a stacking operation while running under the same RB as when it established the ESTAE routine, SDWASRSV contains GPRs 0-15 from the linkage stack entry immediately following the entry that was current when the ESTAE routine was established. Otherwise, SDWASRSV contains GPRs 0-15 from the RB that activated the recovery routine, and the GPRs are the same as they were at the time of the last interruption of that RB that occurred while the RB was unlocked and enabled. Bit SDWAINTF in SDWAXFLG indicates whether the contents of SDWASRSV are from the linkage stack (SDWAINTF is 1) or from an RB (SDWAINTF is 0).

  Note: SDWASRSV is not available for ESTAE-type recovery routines running in AMODE 64. SDWAG64 is used for retry instead of SDWASRSV when RETREGS=64 is specified with SETRP.
- For an ESTAI routine, this field contains zeros.
- For an FRR, this field has the same contents as SDWAGRSV.

If the recovery routine requests a retry, the system might use the contents of this field to load the GPRs for the retry routine. See the RETREGS parameter description in the SETRP macro in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for details. To change the contents of the GPRs for the retry routine, you must make the changes to SDWASRSV and then issue SETRP with RETREGS=YES. You can update the registers directly or with the RUB parameter on SETRP.

SDWAARSV

The contents of this field depend on the type of recovery routine:

- For ESTAE-type recovery routines (except for ESTAI routines): If a program establishes an ESTAE routine, and subsequently performs a stacking operation while running under the same RB as when it established the ESTAE routine, SDWAARSV contains ARs 0-15 from the linkage stack entry immediately following the entry that was current when the ESTAE routine was established. Otherwise, SDWAARSV contains ARs 0-15 from the RB that activated the recovery routine, and the ARs are the same as they were at the time of the last interruption of that RB that occurred while the RB was unlocked and enabled. Bit SDWAINTF in SDWAXFLG indicates whether the contents of SDWAARSV are from the linkage stack (SDWAINTF is 1) or from an RB (SDWAINTF is 0).
- For an ESTAI routine, this field contains zeros.
- For an FRR, this field has the same contents as SDWAARER.

If the recovery routine requests a retry, the system might use the contents of this field to load the ARs for the retry routine. See the RETREGS parameter description in the SETRP macro in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for details. To change the contents of the ARs for the retry routine, you must make the changes in SDWAARSV, and then issue SETRP with RETREGS=YES.

SDWASPID

This field contains the subpool ID of the storage used to obtain the SDWA, for use whenever the retry routine is responsible for freeing the SDWA.

SDWALNTH

This field contains the length, in bytes, of this SDWA, the SDWA extensions, and the variable recording area, for use whenever the retry routine is responsible for freeing the SDWA. (This allows the retry routine to free the extensions along with the SDWA.)

SDWACOMU

The recovery routines can use this 8-byte field to communicate with each other when percolation occurs. The system copies this field from one SDWA to the next on all percolations. When the field contains all zeros, either no information is passed or the system has not been able to pass the information.

Note: Using this field, FRRs can pass information to ESTAE-type recovery routines.

SDWATRAN

This field contains one of the following if a translation exception occurred:

- The valid translation exception address if the SDWATEAV bit is 1.
- The ASID of the address space in which the translation exception occurred if the SDWATEIV bit is 1.

If both the SDWATEAV and SDWATEIV bits are 0, ignore the SDWATRAN field.

SDWATEAR

For translation exceptions that occur in AR mode, this 1-byte field identifies the number of the AR that the program was using when the translation exception occurred.

SDWACLUP

If on, this bit indicates that the recovery routine cannot retry.

SDWAPERC

If on, this bit indicates that a recovery routine of the same type has already percolated for this error. This bit is valid only between similar types of recovery routines; an FRR can tell if an FRR has percolated, and an ESTAE-type recovery routine can tell if an ESTAE-type recovery routine has percolated.

SDWAEAS

If on, this bit indicates that a previous recovery routine provided sufficient diagnostic information pertaining to this error. The recovery routine providing the information is responsible for setting the bit. If the recovery routine issues SDUMPX (or SDUMP), the system sets the bit automatically.

SDWACID

The recovery routine can use this 5-byte field to provide the component ID of the component involved in the error.

SDWASC

The recovery routine can use this 23-byte field to provide the name of the component and a description of the function or subfunction involved in the error.

SDWAMLVL
: The recovery routine can use this 16-byte field to indicate the level of the module involved in the error. The first 8 bytes contains the date (SDWAMDAT) and the second 8 bytes contains the version (SDWAMVRS).

SDWARRL
: The recovery routine can use this 8-byte field to indicate the recovery routine's entry point label.

SDWALCL
: If on, this bit indicates that the system is entering an FRR in LOCAL restricted mode.

SDWALSLV
: The recovery routine can use this 2-byte field to control the linkage stack state upon retry. See “Linkage stack at time of retry” on page 387 for additional information.

SDWAG64
: When running in z/Architecture® mode, this field contains the full 64-bit contents of the general purpose registers at the time of error. It also contains the 64-bit registers to be used for retry if you specify RETREGS=64 on the SETRP macro or turn on the SDWAUPRG and SDWAUP64 bits.

SDWATXG64
: When bits SDWAPCHK and SDWAPTX2 are on, indicating that the program interrupt occurred while within transactional execution, this field contains the full 64-bit contents of the general purpose registers that result from the transaction abort.

SDWATXPSW16
: When bits SDWAPCHK and SDWAPTX2 are on, this field contains the 16-byte PSW that results from the transaction abort.

Special considerations for ESTAE-type recovery routines

This information discusses the following topics related specifically to ESTAE-type recovery:

- RB considerations
- Linkage stack considerations
- Outstanding I/Os at time of failure
- Other considerations specific to ESTAE-type recovery routines
- Using ARRs.

RB considerations

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

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

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

The TOKEN parameter on the ESTAEX or ESTAE macro can help you if your program is running in an environment where a called program is unable to deactivate its own ESTAE-type recovery routine. The TOKEN parameter associates a token with the ESTAE-type recovery routine you are activating. To deactivate (ESTAE or ESTAEX with the 0 parameter) or overlay (ESTAE or ESTAEX with the OV parameter) the recovery routine that was activated with TOKEN, the same token must be specified. All of the more recently activated ESTAE or ESTAEX recovery routines are deactivated at the same time. Using the TOKEN parameter can help in these ways:

- When you attempt to deactivate your ESTAE-type recovery routine, you are sure of deactivating the correct one.
- You deactivate all of the more recently activated ESTAE-type recovery routines, so your program's caller does not have the problem of leftover recovery routines when your program returns control.
- You could activate a dummy ESTAE-type recovery routine with the TOKEN parameter and then deactivate it. The sole purpose of the dummy routine would be to ensure that all leftover ESTAE-type recovery routines are deactivated. Then if your program encounters an error, you are assured that your own ESTAE-type recovery routine gets control.

If a program issues an ESTAE or ESTAEX macro that specifies both the TOKEN parameter and XCTL=YES and then issues the XCTL or XCTLX macro, the token must be passed as part of the parameters to the called routine so that the routine can deactivate the ESTAEX or ESTAE routine.

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

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

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

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

Linkage stack considerations

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

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

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

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

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

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

Outstanding I/Os at the time of failure

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

Note: You need to understand PURGE processing before using this parameter. For information about PURGE processing, see *z/OS DFSMSdfp Advanced Services*.

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

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

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

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

Table 41. Restoring Quiesced Restorable I/O Operations

Parameter on SETRP Macro	RETREGS=NO	RETREGS=YES
FRESDWA=YES	GPR 2 contains the address of the purged I/O restore list (see note below)	Retry routine cannot access the purged I/O restore list.
FRESDWA=NO	GPR 1 contains the address of the SDWA; SDWAFIOB contains the address of the purged I/O restore list	The recovery routine must pass the address of the SDWA to the retry routine; SDWAFIOB contains the address of the purged I/O restore list.

Note: If the system did not provide an SDWA and RETREGS=NO, then GPR 2 contains the address of the purged I/O restore list.

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

Additional considerations specific to ESTAE-type recovery routines

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

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

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

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

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

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

Using ARRs

An ARR provides recovery for a stacking PC routine and receives control if the stacking PC routine encounters an error. An ARR is an ESTAE-type recovery routine. An ARR receives all of the defaults of the ESTAEX macro, with the exception of the TERM parameter. For ARRs, the system uses TERM=YES.

To define an ARR, either:

- Issue the IEAARR macro to establish an ARR to cover a target routine, as described in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.
- Specify the name or address of the routine on the ARR parameter of the ETDEF macro when you define the entry table definition (ETD) for the stacking PC routine; you must also issue the ETCRE macro before the ARR is actually defined to the system. You can use an ARR only for a stacking PC routine that is always entered enabled, unlocked, and in task mode.

You activate an ARR when you issue the PC instruction; you deactivate an ARR when you issue the corresponding PR instruction.

While the stacking PC routine is running, you cannot deactivate its ARR. Therefore, if you do not want an ARR associated with the stacking PC routine, do not code the ARR parameter on the ETDEF macro.

An ARR receives control in 31-bit or 64-bit addressing mode depending on the mode specified via the ETDEF macro or the mode at the time that IEAARR was issued. If it is passed an SDWA, the SDWA is in 31-bit addressable storage.

A stacking PC routine can also have an FRR; however, if the stacking PC routine has an FRR, its ARR gets control only if the FRR percolates. The writer of a stacking PC routine should be aware of the environment in which the routine is called. The system **does not give control to an ARR** in the following circumstances:

- The cross-memory environment required for the ARR (which matches the environment on entry to the stacking PC routine) could not be established.
- Your program is running in SRB mode.
- Your program is running in task mode; prior to issuing the PC instruction to pass control to the stacking PC routine associated with the ARR, you activated one or more FRRs. If the stacking PC routine encounters an error, the most recently activated FRR gets control. If all FRRs percolate, then unless the ARR was defined with ARRCOND=YES, the system records a symptom record in the logrec data set describing the fact that an ARR has been skipped.

If you issue an IEAARR macro to establish an ARR, you can skip the following instructions until the end of this information. See "Setting up, passing, and accessing the parameter area" on page 348 for explanations of passing parameter areas for IEAARR.

If you define the ARR via the ETDEF macro, the stacking PC routine can use the 8-byte modifiable area of the current linkage stack entry to pass parameters to the ARR. To store data into this 8-byte area, the stacking PC routine can use the modify stacked state (MSTA) instruction. The stacking PC routine might, for example, store the address of a footprint area into the first 4 bytes and the ALET identifying the footprint area's address space into the second 4 bytes.

The following example shows how a stacking PC routine can use the 8-byte modifiable area of the linkage stack to pass parameters to an ARR.

```
         LAE    2,PARMAREA      GET ALET AND ADDRESS OF PARMAREA
         EAR    3,2             PUT CONTENTS OF AR2 INTO GPR3
         MSTA   2               PLACE ALET AND ADDRESS OF PARMAREA
*                               INTO LINKAGE STACK
```

Figure 58. Passing Parameters to an ARR

The assembler instructions perform the following functions:

- LAE places the address of PARMAREA into GPR 2 and the ALET that identifies the address space that contains the PARMAREA into AR 2.
- EAR places the contents of AR 2 (the ALET) into GPR 3.
- MSTA places the contents of GPR 2 (the address of PARMAREA) and GPR 3 (the ALET of PARMAREA) into the modifiable area of the linkage stack.

When the ARR receives control, the SDWAPARM field contains the address of a copy of the 8-byte modifiable area of the linkage stack entry established by the stacking PC routine associated with the ARR. For example, suppose PGM1 uses a stacking PC to call PCRTN1. After PCRTN1 gets control, it encounters an error and control passes to the ARR associated with the stacking PC routine. At this time, SDWAPARM contains the address of a copy of the modifiable area in the linkage stack entry associated with the stacking PC routine. Note that SDWAPARM does not necessarily contain the address of a copy of the current entry. Between the time it received control and encountered the error, PCRTN1 could have added an entry to the linkage stack through a BAKR instruction. This entry would be the current entry; SDWAPARM contains the address of a copy of a previous entry.

If there is no SDWA, the ARR receives the copy of the 8-byte area in GPR/AR 2: GPR 2 contains the first 4 bytes, and AR 2 contains the second 4 bytes.

If there is no SDWA when the recovery routine requests a retry, the system uses the linkage stack entry that was current when the ARR's stacking PC routine was entered, and from that entry, provides a copy of the current contents of the 8-byte modifiable area in GPR/AR 1.

You have the option to allow or prevent asynchronous interrupts from interrupting an ARR while it is running. To allow asynchronous interrupts, specify ASYNCH=YES on the ETDEF macro. To prevent asynchronous interrupts from interrupting the ARR, specify ASYNCH=NO. If the ARR uses any supervisor services that depend on an asynchronous interrupt for successful completion, specify ASYNCH=YES.

You also have the option to allow cancel or detach processing to interrupt an ARR while it is running or to prevent this interrupt. To allow the interrupt, specify CANCEL=YES on the ETDEF macro. To prevent the interrupt, specify CANCEL=NO. If you specify CANCEL=NO, and a cancel or detach is attempted while the ARR is running, the ARR completes running before the system begins cancel or detach processing. The routine that issues ETDEF cannot use CANCEL=NO unless the stacking PC routine to be associated with the ARR runs in supervisor state or with PSW key 0 - 7.

One ARR can serve multiple stacking PC routines. In this case, each stacking PC routine must not only store information about where in its processing the stacking PC routine is, but must also store information that identifies which stacking PC routine it is.

Special considerations for FRRs

Consider the following when writing an FRR:

FRR stack: When you define an FRR, the SETFRR macro creates an entry in a system area called an FRR stack. The size of the FRR stack satisfies the recovery needs of the system. If additional FRRs placed on the stack cause the size to be exceeded, the routine issuing the SETFRR macro ends abnormally. User-written routines outside of the system may add up to two FRRs to the FRR stack. All remaining stack entries are reserved for the system. If a routine attempts to add an entry to the stack when the stack is full, the system issues an X'07D' abend.

SDWA: The system always gives control to FRRs with an SDWA. If the system cannot supply an SDWA, the system bypasses the FRR, goes to the next FRR, and again attempts to supply an SDWA. When no FRRs remain, the system gives control to any ESTAE-type recovery routines that are available. When an FRR receives control in AMODE 64 the SDWA may be in 31-bit addressable storage.

AMODE 31 users of SETFRR can specify SDWALOC31=YES to indicate that their FRR can tolerate an SDWA in 31-bit addressable storage. Considering that 31-bit storage is less likely to be constrained than 24-bit storage and RTM skips FRRs for which it can not obtain an SDWA, use SDWALOC31=YES whenever possible for AMODE 31 FRRs.

Locks: When the system invokes an FRR, the FRR runs with the locks (and the enablement implied by those locks) that were held at the time of the error, plus an additional CPU lock obtained for the system if the FRR was holding any CPU locks, unless a previous FRR does one of the following:

- Deletes one or more of the locks
- Requests the system to delete one or more of the locks using the FRELOCK parameter on the SETRP macro.

The FRR should take the following action regarding locks held when the FRR is entered:

- The FRR should issue SETRP with the FRELOCK parameter. This indicates that the system is to free the specified locks if the FRR percolates.
- Assuming retry is not prohibited, if the FRR retries, it should request that the system free locks prior to entering the retry routine by specifying the following on SETRP:
  - The FRELOCK parameter, specifying the locks to be freed.
  - The FRLKRTY=YES parameter.
- If the FRR needs to do some processing that cannot be done with the locks that were held at time of entry, keep the following in mind:
  - If entered holding one or more CPU locks (that is, the FRR is entered disabled), then the FRR must not release **all** of the CPU locks (**it must remain disabled by retaining at least one CPU lock**). Freeing the last CPU lock can cause different units of work to use the same SDWA simultaneously.
  - If entered holding the CMS lock, the FRR can release that lock.
  - If entered holding the LOCAL lock or a CML lock, the FRR can release that lock if either of the following is true:
    - The FRR is in SRB mode.
    - The FRR is in task mode and is an EUT FRR.

Deactivating an FRR: When an FRR wants to deactivate itself, the FRR should issue SETRP with REMREC=YES. FRRs can also issue SETFRR D, but must exercise caution when doing so. The FRR should not call any other program or service after issuing SETFRR D because of the following:

- When the FRR issues SETFRR D, it removes itself from the FRR stack.
- If the FRR then calls a program or service that sets another FRR, that FRR goes on the FRR stack.
- The system thinks the new FRR is the original FRR.
- In the event the called program or service encounters an error, the system does not give control to the new FRR because it appears to be the original FRR.

For example, suppose the FRR issues SETFRR D, and then issues the SDUMPX macro. SDUMPX processing places another FRR on the FRR stack, in the location of the original FRR. If SDUMPX processing encounters an error, the system does not give control to its FRR because the system thinks it is the original FRR, to which control was already given.

Thus, **IBM recommends** that you issue SETRP with REMREC=YES rather than issuing SETFRR D within an FRR. If the FRR specifies SETRP REMREC=YES, the system deactivates the FRR after the FRR gives up control, thus removing the entry from the FRR stack at a safe time.

Understanding the recovery environment

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

- **Register contents.**

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

  – When the error occurs

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

  – On entry to and return from the recovery routine

    See "Register contents on entry to a recovery routine" on page 366 and "Register contents on return from a recovery routine" on page 369 for details.

  – On entry to the retry routine

    See "Register contents on entry to a retry routine" on page 369 for details.

- **All other environmental factors.**

  The other environmental factors important in a recovery environment are:

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

  – SDWA storage key

  – Dispatchable unit mode

  – Cross memory mode

  – AMODE

  – ASC mode

  – Interrupt status

  – Dispatchable unit access list (DU-AL)

  – Locks

  – Program mask

  – Condition of the linkage stack

  – Authorization index (AX)

  – Extended authorization index (EAX)

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

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

Register contents

This information describes register contents for the following:

- On entry to a recovery routine
- On return from a recovery routine (see "Register contents on return from a recovery routine" on page 369)
- On entry to a retry routine.

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

Table 42. Where to Find Register Content Information

Registers Described For:	**Table Number:**
ESTAE-type recovery routine with an SDWA	Table 43
ESTAE-type recovery routine without an SDWA	Table 44 on page 367
FRR	Table 45 on page 368
Retry from an ESTAE-type recovery routine without an SDWA	Table 46 on page 370
Retry from an ESTAE-type recovery routine with an SDWA, RETREGS=NO, and FRESDWA=NO	Table 48 on page 371
Retry from an ESTAE-type recovery routine with an SDWA, RETREGS=NO, and FRESDWA=YES	Table 50 on page 371
Retry from an ESTAE-type recovery routine with an SDWA and RETREGS=YES	Table 52 on page 372
Retry from an ESTAE-type recovery routine with an SDWA and RETREGS=64 in z/Architecture mode	Table 54 on page 372
Retry from an FRR	Table 56 on page 373
Retry from an FRR with RETREGS=64 in z/Architecture mode	Table 58 on page 373

Register contents on entry to a recovery routine

The register contents on entry to a recovery routine are different depending on whether you have an ESTAE-type recovery routine or an FRR, and depending on whether the system supplied an SDWA. The possibilities are:

- ESTAE-type recovery routine with an SDWA
- ESTAE-type recovery routine without an SDWA
- FRR (FRRs always get control with an SDWA).

The following tables describe the register contents on entry to the recovery routine for each of the above situations.

Table 43. Register Contents—ESTAE-Type Recovery Routine With an SDWA

Register	**Contents**
	General Purpose Registers
GPR 0	A code indicating the type of I/O processing performed: **0** Active I/O has been quiesced and is restorable. **4** Active I/O has been halted and is not restorable. **8** No I/O was active when the abend occurred. **16 (X'10')** No I/O processing was performed.
GPR 1	Address of the SDWA.

Table 43. Register Contents—ESTAE-Type Recovery Routine With an SDWA (continued)

Register	Contents
GPR 2	One of the following: • For ARRs, a copy of the first 4 bytes of the 8-byte modifiable area of the linkage stack. • If you specified the PARM parameter on ESTAE or ESTAEX, the address of the user-supplied parameter area. • For ESTAI issued via ATTACHX, the address of the user-supplied parameter area. Note this is a 31-bit address for ESTAI issued in either AMODE 31 or AMODE 64. • If you issued FESTAE without the PARAM parameter, the address of the 24-byte parameter area in the SVRB (RBFEPARM). • If you issued ESTAE, ESTAEX, or ATTACHX without the PARAM parameter, zero. • For IEAARR issued in AMODE 31, the 31-bit address of the parameter area specified on the ARRPARAMPTR parameter of IEAARR. • For IEAARR issued in AMODE 64, the 64-bit address of the parameter area specified on the ARRPARAMPTR64 parameter of IEAARR.
GPRs 3 - 12	Do not contain any information for use by the routine.
GPR 13	Address of a 144-byte register save area.
GPR 14	Return address to the system.
GPR 15	Entry point address of the ESTAE-type recovery routine except for ESTAEX issued in AMODE 64 in which the low order bit is set on.
	Access Registers
ARs 0 - 1	Zero
AR 2	One of the following: • For ARRs, a copy of the second 4 bytes of the 8-byte modifiable area of the linkage stack. • If you issued the ESTAEX macro in AR ASC mode and not AMODE 64, an ALET that qualifies the address in GPR 2. • Otherwise, this register does not contain any information for use by the routine.
ARs 3 - 15	Zero.

Table 44. Register Contents—ESTAE-Type Recovery Routine Without an SDWA

Register	Contents
	General Purpose Registers
GPR 0	12 (X'0C'). The system could not obtain an SDWA.
GPR 1	Completion code in bytes 1-3. The system completion code appears in the first 12 bits, and the user completion code appears in the second 12 bits.

Table 44. Register Contents—ESTAE-Type Recovery Routine Without an SDWA (continued)

Register	Contents
GPR 2	One of the following: • For ARRs, a copy of the first 4 bytes of the 8-byte modifiable area of the linkage stack. • If you specified the PARAM parameter on ESTAE, ESTAEX, or FESTAE, the address of the user-supplied parameter area. • For ESTAI issued via ATTACHX, the address of the user-supplied parameter area. Note this is a 31-bit address for ESTAI issued in either AMODE31 or AMODE64. • If you issued FESTAE without the PARM parameter, the address of the 24-byte parameter area in the SVRB (RBFEPARM). • If you issued ESTAE, ESTAEX, or ATTACHX without the PARAM parameter, zero. • For IEAARR issued in AMODE 31, the 31-bit address of the parameter area specified on the ARRPARAMPTR parameter of IEAARR. • For IEAARR issued in AMODE 64, the 64-bit address of the parameter area specified on the ARRPARAMPTR64 parameter of IEAARR.
GPRs 3 - 13	Do not contain any information for use by the routine. **Note:** When the system does not provide an SDWA, GPR 13 **does not** contain the address of a 144-byte save area. In this case, your ESTAE-type recovery routine must save the address from GPR 14 and use it as the return address to the system.
GPR 14	Return address to the system.
GPR 15	Entry point address of the ESTAE-type recovery routine.
	Access Registers
ARs 0 - 1	Zero
AR 2	One of the following: • For ARRs, a copy of the second 4 bytes of the 8-byte modifiable area of the linkage stack. • If you issued the ESTAEX macro in AR ASC mode, an ALET that qualifies the address in GPR 2. • Otherwise, this register does not contain any information for use by the routine.
ARs 3 - 15	Zero.

Table 45. Register Contents—FRR

Register	Contents
	General Purpose Registers
GPR 0	Address of a 304-byte work area for the FRR. This area has been cleared to zeros.
GPR 1	Address of the SDWA. **Note:** When an FRR receives control in AMODE 64 the SDWA may be in 31-bit addressable storage.
GPR 2	Contains the address of the 24-byte parameter area returned by the SETFRR macro when you specified PARMAD.
GPRs 3 - 13	Do not contain any information for use by the routine.
GPR 14	Return address to the system.

Table 45. Register Contents—FRR (continued)

Register	Contents
GPR 15	Address of the FRR.
	Access Registers
ARs 0 - 15	Zero

Register contents on return from a recovery routine

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

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

Return Code
Meaning

0 The recovery routine requests percolation.

4 The recovery routine requests a retry. **The recovery routine must then place the address of the retry routine in GPR 0.**

16 (X'10')
Valid only for an ESTAI recovery routine. The system should not give control to any further ESTAI routines, and should abnormally end the task.

Register contents on entry to a retry routine

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

- Whether this is a retry from an ESTAE-type recovery routine or an FRR.
- Whether an SDWA is present.
- If an SDWA is present, what the recovery routine specifies on the SETRP macro.

Retry from an ESTAE-type recovery routine: When an ESTAE-type recovery routine attempts a retry, the contents of the registers on entry to the retry routine depend on whether an SDWA is present, and depend on what the recovery routine specifies on SETRP. The parameters on SETRP that affect register contents on entry to the retry routine from an ESTAE-type recovery routine are the following:

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

  If you specify RETREGS=YES, GPRs are restored from SDWASRSV, and ARs are restored from SDWAARSV. If you specify RETREGS=YES,RUB, you can manipulate the contents of SDWASRSV to whatever you wish the GPRs to be before they are restored. Or, you can directly manipulate the contents of both SDWASRSV and SDWAARSV. When you specify RETREGS=YES and are running in z/Architecture mode, the upper halves of the 64–bit registers at retry will contain the upper halves of the 64–bit registers from the time of error.

  If you are running in z/Architecture mode and specify RETREGS=64, the 64–bit GPRs at retry are restored from SDWAG64 and the ARs are restored from SDWAARSV.

  See the description of the SETRP macro in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for complete details.

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

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

Table 46. Register Contents—Retry from an ESTAE-Type Recovery Routine Without an SDWA (General Purpose Registers)

Register	Contents
GPR 0	12 (X'0C').
GPR 1	For ARRs, a copy of the current contents of the first 4 bytes of the 8-byte modifiable area in the linkage stack entry that was current when the ARR''s stacking PC was entered. For IEAARR issued in AMODE 31, the 31-bit address of the parameter area specified on the ARRPARAMPTR parameter of IEAARR. For IEAARR issued in AMODE 64, the 64-bit address of the parameter area specified on the ARRPARAMPTR64 parameter of IEAARR. For other ESTAE-type recovery routines where the PARAM parameter was specified, the address of the user-supplied parameter area. Note that when ESTAEX was issued in AMODE 64, GPR 1 contains a 64-bit value. For other ESTAE-type routines where the PARAM parameter was not specified, zero.
GPR 2	Address of the purged I/O restore list if I/O was quiesced and is restorable; otherwise, zero.
GPRs 3 - 14	Do not contain any information for use by the routine.
GPR 15	Entry point address of the retry routine, except for when ESTAEX was issued in AMODE 64, in which case the low order bit is set on.

Table 47. Register Contents—Retry from an ESTAE-Type Recovery Routine Without an SDWA (Access Registers)

Register	Contents
AR 0	Zero.
AR 1	One of the following: • If you issued the ESTAEX macro in AR ASC mode and not AMODE 64, an ALET that qualifies the address in GPR 1. • For ARRs, the system uses the linkage stack entry that was current when the ARR's stacking PC routine was entered, and from that entry, provides a copy of the current contents of the second 4 bytes of the 8-byte modifiable area. • Otherwise, this register does not contain any information for use by the routine.
ARs 2 - 13	Do not contain any information for use by the routine.
ARs 14 - 15	Zero.

Table 48. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=NO (General Purpose Registers)

Register	Contents
GPR 0	Zero.
GPR 1	Address of the SDWA.
GPRs 2 - 14	Do not contain any information for use by the routine.
GPR 15	Entry point address of the retry routine, except for when ESTAEX was issued in AMODE 64, in which case the low order bit is set on.

Table 49. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=NO (Access Registers)

Register	Contents
ARs 0 - 1	Zero.
ARs 2 - 13	Do not contain any information for use by the routine.
ARs 14 - 15	Zero.

Table 50. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=YES (General Purpose Registers)

Register	Contents
GPR 0	20 (X'14').
GPR 1	For ARRs, a copy of the current contents of the first 4 bytes of the 8-byte modifiable area in the linkage stack entry that was current when the ARR''s stacking PC was entered. For IEAARR issued in AMODE 31, the 31-bit address of the parameter area specified on the ARRPARAMPTR parameter of IEAARR. For IEAARR issued in AMODE 64, the 64-bit address of the parameter area specified on the ARRPARAMPTR64 parameter of IEAARR. For other ESTAE-type recovery routines where the PARAM parameter was specified, the address of the user-supplied parameter area. Note that when ESTAEX was issued in AMODE 64, GPR 1 contains a 64-bit value. For other ESTAE-type recovery routines where the PARAM parameter was not specified, zero.
GPR 2	Address of the purged I/O restore list, if I/O was quiesced and is restorable; otherwise, zero.
GPRs 3 - 14	Do not contain any information for use by the routine.
GPR 15	Entry point address of the retry routine, except for when ESTAEX was issued in AMODE 64, in which case the low order bit is set on.

Table 51. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=YES (Access Registers)

Register	Contents
AR 0	Zero.

Table 51. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA, RETREGS=NO, and FRESDWA=YES (Access Registers) (continued)

Register	Contents
AR 1	One of the following: • If you issued the ESTAEX macro in AR ASC mode and not AMODE 64, an ALET that qualifies the address in GPR 1. • For ARRs, the system uses the linkage stack entry that was current when the ARR's stacking PC routine was entered, and from that entry, provides a copy of the current contents of the second 4 bytes of the 8-byte modifiable area. • Otherwise, this register does not contain any information for use by the routine.
ARs 2 - 13	Do not contain any information for use by the routine.
ARs 14 - 15	Zero.

Table 52. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA and RETREGS=YES (General Purpose Registers)

Register	Contents
GPRs 0 - 15	Restored from SDWASRSV, regardless of whether the recovery routine specified FRESDWA=NO or FRESDWA=YES. Note that register 15 does not contain the entry point address of the retry routine unless the recovery routine sets it up that way.

Table 53. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA and RETREGS=YES (Access Registers)

Register	Contents
ARs 0 - 15	Restored from SDWAARSV, regardless of whether the recovery routine specified FRESDWA=NO or FRESDWA=YES.

Table 54. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA and RETREGS=64 in z/Architecture mode (General Purpose Registers)

Register	Contents
GPRs 0 - 15	Restored from SDWAG64, regardless of whether the recovery routine specified FRESDWA=NO or FRESDWA=YES. Note that register 15 does not contain the entry point address of the retry routine unless the recovery routine sets it up that way.

Table 55. Register Contents—Retry from an ESTAE-Type Recovery Routine With an SDWA and RETREGS=64 in z/Architecture mode (Access Registers)

Register	Contents
ARs 0 - 15	Restored from SDWAARSV, regardless of whether the recovery routine specified FRESDWA=NO or FRESDWA=YES.

Retry from an FRR: The following table describes register contents on entry to a retry routine from an FRR. Note that an FRR can use RETREGS=YES,RUB to update the SDWASRSV, or can directly manipulate the contents of both the SDWASRSV and the SDWAARSV.

If you are running in z/Architecture mode and do not specify RETREGS=64, the upper halves of the 64-bit GPRs at retry will contain the upper halves of the 64-bit registers from the time of error.

Table 56. Register Contents—Retry from an FRR (General Purpose Registers)

Register	Contents
GPRs 0 - 14	Restored from SDWASRSV.
GPR 15	Entry point for the retry routine if the FRR specifies RETRY15=NO, or takes the default; restored from SDWASRSV if the FRR specifies RETRY15=YES.

Table 57. Register Contents—Retry from an FRR (Access Registers)

Register	Contents
ARs 0 - 14	Restored from SDWAARSV.
AR 15	Zero if the FRR specifies RETRY15=NO, or takes the default; restored from SDWAARSV if the FRR specifies RETRY15=YES.

Table 58. Register Contents—Retry from an FRR with RETREGS=64 in z/Architecture mode (General Purpose Registers)

Register	Contents
GPRs 0 - 14	Restored from SDWAG64.
GPR 15	Entry point for the retry routine if the FRR specifies RETRY15=NO, or takes the default; restored from SDWASRSV if the FRR specifies RETRY15=YES. When GPR 15 contains the entry point of the retry routine, the upper half of GPR 15 is set to zero.

Table 59. Register Contents—Retry from an FRR with RETREGS=64 in z/Architecture mode (Access Registers)

Register	Contents
ARs 0 - 14	Restored from SDWAARSV.
AR 15	Zero if the FRR specifies RETRY15=NO, or takes the default; restored from SDWAARSV if the FRR specifies RETRY15=YES.

Other environmental factors in recovery

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

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

- AX
- EAX

These environmental factors differ depending on whether the routine is:

- An ESTAE-type recovery routine
- A retry routine getting control from an ESTAE-type recovery routine
- An FRR
- A retry routine getting control from an FRR.

Environment on entry to an ESTAE-type recovery routine

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

Authorization:

Environmental factor	Requirement
Problem or supervisor state	An ESTAE-type recovery routine is entered in the state, either supervisor or problem state, that existed at the time the recovery routine was defined.
PSW key	An ESTAE-type recovery routine is entered with PSW key 0 whenever the routine was defined with a PSW key less than 8. Otherwise, it is entered with the PSW key that existed at the time it was defined.
PKM	ESTAE-type recovery routines defined by the ESTAE, ESTAEX, or ATTACHX macros are entered with the PKM that existed at the time the macro was issued. FESTAE routines are entered with the PKM that existed at the time of error. ARRs are entered with the PKM that existed at the time their stacking PC routine was entered.

SDWA storage key: An ESTAE-type recovery routine activated under PSW key 0 - 7 receives an SDWA in storage key 0.

An ESTAE-type recovery routine activated under PSW key 8 - F receives an SDWA as follows:

- Whether or not the recovery routine is in cross memory mode, the SDWA is in the same storage key as the TCB key at the time the related task made the first storage request from subpool 0.
- If the recovery routine is in cross memory mode, the system obtains the SDWA from the primary address space in which the task that owns the cross memory resources is running; that task is the one whose address is in ASCBXTCB.

For information about subpools and storage keys, see “Selecting the right subpool for your virtual storage request” on page 217.

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

Cross memory mode: A recovery routine defined through the ESTAE macro, the FESTAE macro, or the ESTAI parameter on ATTACHX has PASN=HASN=SASN.

A recovery routine defined through the ESTAEX macro has the PASN and SASN that existed when the ESTAEX macro was issued.

ARRs have the PASN and SASN that existed when the stacking PC routine was entered.

AMODE: ESTAE-type recovery exits receive control in the AMODE that was current at the time-of-set (time-of-PC AMODE for ARRs) with the following exceptions:

- ARR, IEAARR, and ESTAEX exits receive control in AMODE 31 instead of AMODE 24 when established for AMODE 24 programs.

ASC mode: A recovery routine defined through the ESTAE macro or the FESTAE macro is entered in primary ASC mode.

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

ARRs are entered in the ASC mode that existed at the time the stacking PC routine was entered.

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

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

Locks: All locks are freed before the first ESTAE-type recovery routine gets control.

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

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

AX: The AX for ESTAE-type recovery routines is the AX for the current PASN of the recovery routine. The system does not modify the AX value during recovery processing. Refer to the information on cross memory in *z/OS MVS Programming: Extended Addressability Guide* for more information about the AX.

EAX: The EAX for recovery routines defined through the ESTAE macro, the ESTAEX macro, or the ESTAI parameter on ATTACHX is the same as when the macro was issued.

For FESTAE routines, the EAX is the same as when the SVC routine that defined the FESTAE routine was entered.

For ARRs, the EAX is the same as when the stacking PC routine was entered.

Refer to the information on access registers in *z/OS MVS Programming: Extended Addressability Guide* for more information about the EAX.

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

1. Check the STCBRMET field in the STCB; if the bit is on, the ESTAI routine is running **after** a resource manager has ended abnormally and its recovery routines have not retried. In this situation, the ESTAI routine does not need to hold a lock to check the STCBRMET field. For the mapping of the STCB, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).
2. Do as little processing as possible, and nothing that might depend on a resource that might have been cleaned up already.
3. Do **not** request to retry. The system will not allow a retry in this situation.

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

Environment on entry to a retry routine from an ESTAE-type recovery routine

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

Authorization:

Environmental factor	Requirement
Problem or supervisor state	If the recovery routine was defined by an ESTAE or ESTAEX macro, the retry routine is entered in the state that existed when the macro was issued. If the recovery routine was defined by a FESTAE macro, the retry routine is entered in supervisor state. For ARRs, the retry routine is entered in the state that existed when the stacking PC routine was entered. If the recovery routine was defined by the ESTAI parameter of the ATTACHX macro, the retry routine is entered in supervisor state if, and only if, the RBOPSW of the retry RB is in supervisor state and the mainline routine was authorized at the time of the error. Otherwise, the retry routine is entered in problem state. The mainline routine is considered to be authorized at the time of the error when at least one of the following is true: • The program is APF-authorized. • The task in error has a protect key which is less than 8. • All RBs for the task in error run in supervisor state.
PSW key	If the recovery routine was defined by an ESTAE, ESTAEX, or ESTAEX macro, the retry routine is entered with the same PSW key that existed when the macro was issued. If the recovery routine was defined by the ESTAI parameter of the ATTACHX macro, the retry routine is entered with the same PSW key as the one in RBOPSW of the retry RB, if the RBOPSW of the retry RB when one of the following is true (otherwise, the PSW key of the retry routine is that of the task in error): • The main routine was authorized (as defined earlier under “Problem or Supervisor State”) at the time of the error. • The RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem state, and the PKM of that RB does not have authority to keys less than 8. For ARRs, the retry routine is entered with the same PSW key that existed when the stacking PC routine was entered.

Environmental factor	Requirement
PKM	If the recovery routine was defined through the ESTAE or ESTAEX macro, the retry routine is entered with the PKM that existed when the macro was issued. If the recovery routine was defined by the FESTAE macro, the retry routine is entered with the PKM that existed at the time of the error. If the recovery routine was defined through the ESTAI parameter of the ATTACHX macro, the retry routine is entered with the PKM from the retry RB only if one of the following is true (otherwise, the PKM of the retry routine only has authority that is equivalent to that of the task in error): • The main routine was authorized (as defined earlier under "Problem or Supervisor State") at the time of the error. • The RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem state, and the PKM of that RB does not have authority to keys less than 8. For ARRs, the retry routine is entered with the PKM at the time the PC routine was entered.

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

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

Cross memory mode: If the recovery routine was defined through the ESTAE macro, the FESTAE macro, or the ESTAI parameter on ATTACHX, the retry routine is entered with PASN=HASN=SASN.

If the recovery routine was defined through the ESTAEX macro, the retry routine is entered with the PASN and SASN that existed when the macro was issued.

For ARRs, the retry routine is entered with the PASN and SASN that existed when the stacking PC routine was entered.

AMODE: Retry routines are entered in the same addressing mode that existed when the recovery routine was entered, unless the SETRP RETRYAMODE= is used by the recovery routine. See the description of RETRYAMODE= parameter on the SETRP information of the *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

ASC mode: For recovery routines defined through the ESTAE or FESTAE macro, the retry routine is entered in primary ASC mode.

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

For ARRs, the retry routine is entered in the ASC mode at the time the stacking PC routine was entered.

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

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

Locks: The retry routine is entered with no locks held. If the ESTAE-type recovery routine obtains any locks, it must free those locks before returning to the system. Otherwise, the system recognizes this as an error, and the retry routine does not get control.

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

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

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

For FESTAE routines, on entry to the retry routine, the current linkage stack entry is the same as it was at the time the SVC routine was entered.

For ARRs, on entry to the retry routine, the current linkage stack entry is the same as it was at the time the PC routine was entered, unless the recovery routine has set the SDWALSLV field.

AX: The retry routine is entered with the AX for the current PASN of the retry routine. The system does not modify the AX value during recovery processing.

EAX: For recovery routines defined through the ESTAE or ESTAEX macro, the EAX for the retry routine is the same as when the macro was issued.

For ESTAI routines, the EAX for the retry routine is the EAX at the time the selected retry RB was entered.

For FESTAE routines, the EAX for the retry routine is the same as when the SVC routine was entered.

For ARRs, the EAX for the retry routine is the same as when the PC routine was entered.

Summary of environment on entry to an ESTAE-type recovery routine and its retry routine

Table 60 summarizes some of the environmental factors for ESTAE-type recovery routines under different conditions. Specifically, the table lists the status information of:

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

Table 60. Environments of ESTAE-Type Recovery Routines and their Retry Routines

Type of Recovery	When macro was issued	At entry to recovery routine	At entry to retry routine
ESTAE	ASC mode=primary PASN=HASN=SASN	ASC mode=primary PASN=HASN=SASN Linkage stack at time of error (see Note 1 on page 381) EAX at time macro was issued PKM at time macro was issued	ASC mode=primary PASN=HASN=SASN Linkage stack at time macro was issued (see Note 2 on page 381) EAX at time macro was issued PKM at time macro was issued
ESTAEX or IEAARR	ASC mode=primary or AR PASN=HASN=SASN or any PASN, any SASN, any HASN	ASC mode at time macro was issued PASN and SASN at time macro was issued Linkage stack at time of error (see Note 1 on page 381) EAX at time macro was issued PKM at time macro was issued	ASC mode at time macro was issued PASN and SASN at time macro was issued Linkage stack at time macro was issued (see Note 2 on page 381) EAX at time macro was issued PKM at time macro was issued
ARR	N/A	ASC mode at time stacking PC routine was entered PASN and SASN at time stacking PC routine was entered Linkable stack at time of error (see Note 1 on page 381) EAX at time stacking PC routine was entered PKM at time stacking PC routine was entered	ASC mode at time stacking PC routine was entered PASN and SASN at time stacking PC routine was entered Linkage stack at time stacking PC routine was entered (see Note 2 on page 381) EAX at time stacking PC routine was entered PKM at time stacking PC routine was entered
FESTAE	ASC mode=primary PASN=HASN=SASN	ASC mode=primary PASN=HASN=SASN Linkage stack at time of error (see Note 1 on page 381) EAX at time SVC routine was entered PKM at time of error	ASC mode=primary PASN=HASN=SASN Linkage stack at time SVC routine was entered EAX at time SVC routine was entered PKM at time of error

Table 60. Environments of ESTAE-Type Recovery Routines and their Retry Routines (continued)

Type of Recovery	When macro was issued	At entry to recovery routine	At entry to retry routine
ESTAI (through ATTACHX)	ASC mode=primary or AR PASN=HASN=SASN	ASC mode at time macro was issued PASN=HASN=SASN Linkage stack at time of error (see Note 1) EAX at time macro the selected retry RB was entered (see Note 2) EAX at time the macro was issued PKM at time macro was issued For possible environment restrictions, see “Restricted environments” on page 376	ASC mode at time macro was issued PASN=HASN=SASN Linkage stack at time the selected retry RB was entered (see Note 2) EAX at time the selected retry RBG was entered

Note:

1. If a previous recovery routine added entries through a PC or BAKR instruction and did not remove them, the linkage stack presented to the recovery routine might have additional entries beyond what was the current entry at the time of the error.
2. When the recovery routine set the SDWALSLV field, the linkage stack presented to the retry routine might have additional entries beyond what was current at the time the recovery routine was activated.
3. At time of entry to the recovery routine, the AMODE is the same as the time of invocation, except for ESTAEX, ARR, and IEAARR routines. ESTAEX and IEAARR routines that were established in AMODE 64 receive control in AMODE 64, otherwise ESTAEX, ARR, and IEAARR routines always receive control in AMODE 31.
4. The AMODE at the retry point is the same as the AMODE on entry to the recovery routine, unless the SETRP RETRYAMODE= parameter specifies a specific retry AMODE.
5. An ESTAE-type recovery routine is entered with PSW key 0 whenever it was defined with a PSW key less than 8. Otherwise, it is entered with the PSW key that existed at the time it was defined.
6. The AX for the recovery routine or the retry routine is always the AX for the current PASN of the recovery routine or the retry routine, respectively. The system does not modify the AX value during recovery processing.

Environment on entry to an FRR

The following is a description of each environmental factor on entry to an FRR. In addition to the factors described for ESTAE-type recovery routines, this information includes a description of restricted environments, which applies only to FRRs. See “Restricted environments” on page 384 for more information.

Authorization:

Environmental factor	Requirement
Problem or supervisor state	All FRRs receive control in supervisor state.
PSW key	All FRRs receive control in PSW key 0.

Environmental factor	Requirement
PKM	For FRRs defined through the SETFRR macro with the MODE=FULLXM or MODE=PRIMARY parameters, the PKM on entry to the FRR is the PKM at the time the SETFRR macro was issued. For FRRs defined through the SETFRR macro with the MODE=HOME parameter, the PKM on entry to the FRR is as follows: • If the FRR is in SRB mode, the PKM at the time of the error. • If the FRR is in task mode, the PKM that can be constructed from the storage protection key in the TCB of the currently active task, unless a DAT error occurs. If a DAT error occurs, the PKM is the PKM at the time of the error.

SDWA storage key: FRRs always receive an SDWA in storage key 0.

Dispatchable unit mode: FRRs are entered in the dispatchable unit mode that existed at the time of error.

Cross memory mode: The cross memory environment for an FRR depends on the values coded for the MODE parameter in the SETFRR macro that defined the FRR.

Specifying HOME, PRIMARY, or FULLXM for the MODE parameter of the SETFRR macro indicates to the system the normal or expected addressing environment of the FRR.

MODE=HOME
If you specify MODE=HOME or omit the MODE parameter, the FRR gets control with PASN=HASN=SASN.

MODE=PRIMARY
If you code MODE=PRIMARY, the FRR gets control with the PASN and SASN both the same as the PASN that existed when the SETFRR macro was issued.

MODE=FULLXM
If you specify MODE=FULLXM, the FRR gets control in the cross memory environment that existed when the SETFRR macro was issued. That is, the PASN and the SASN are the same as those that existed when the SETFRR macro was issued.

If the system cannot enter an FRR with its normal addressing environment established as defined by the MODE parameter on the SETFRR macro, the system bypasses the FRR and percolates to the next FRR on the FRR stack, unless the FRR was defined to run in a restricted addressing environment (also specified by the MODE parameter). See “Restricted environments” on page 384 for more information about FRRs running in restricted environments.

AMODE: The addressing mode for FRRs is the addressing mode that existed at the time the SETFRR macro was issued.

ASC mode: The ASC mode on entry to an FRR depends on the ASC mode when the SETFRR macro was issued, and on the MODE specification on the SETFRR macro. See Table 61 for the ASC mode on entry to an FRR.

Interrupt status: An FRR gets control and is entered disabled if, at the time of the error, the mainline routine is disabled. **Any FRR entered disabled must remain disabled.**

DU-AL: FRRs receive control with the DU-AL that was current at the time of the error, as modified by any previous FRRs. The system does not modify the contents of the DU-AL during recovery processing.

Locks: FRRs run with the locks (and the enablement implied by those locks) that were held at the time of the error, plus an additional CPU lock obtained for the system if the FRR was holding any CPU locks, unless a previous FRR does one of the following:

- Deletes one or more of the locks
- Requests the system to delete one or more of the locks using the FRELOCK parameter on the SETRP macro.

Program mask: On entry to an FRR, the program mask is zero.

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

AX: The AX for FRRs is the AX for the current PASN of the recovery routine. The system does not modify the AX value during recovery processing. Refer to the information on cross memory in *z/OS MVS Programming: Extended Addressability Guide* for more information about the AX.

EAX: The EAX for FRRs is the same as when the SETFRR macro was issued. Refer to the information on access registers in *z/OS MVS Programming: Extended Addressability Guide* for more information about the EAX.

Summary of environment on entry to an FRR: Table 61 summarizes some of the environmental factors on entry to FRRs for three options on the MODE parameter:

Table 61. Environment on Entry to FRRs

	MODE=FULLXM	**MODE=PRIMARY**	**MODE=HOME**

SETFRR issued in primary ASC mode	ASC mode=primary
	PASN=PASN at the time SETFRR was issued
	SASN=SASN at the time SETFRR was issued
	PKM=PKM at the time SETFRR was issued
		ASC mode=primary
		PASN=PASN at the time SETFRR was issued
		SASN=PASN at the time SETFRR was issued
		PKM=PKM at the time SETFRR was issued
			ASC mode=primary
			PASN=HASN=SASN
			PKM depends on the dispatchable unit mode, as follows:
			• For task mode, see Note 1 on page 384.
			• For SRB mode, PKM=PKM at time of error

Table 61. Environment on Entry to FRRs (continued)

	MODE=FULLXM	MODE=PRIMARY	MODE=HOME

SETFRR issued in secondary ASC mode	ASC mode=secondary
	PASN=PASN at the time SETFRR was issued
	SASN=SASN at the time SETFRR was issued
	PKM=PKM at the time SETFRR was issued
		ASC mode=primary
		PASN=PASN at the time SETFRR was issued
		SASN=PASN at the time SETFRR was issued
		PKM=PKM at the time SETFRR was issued
			ASC mode=primary
			PASN=HASN=SASN
			PKM depends on the dispatchable unit mode, as follows:
			• For task mode, see Note 1.
			• For SRB mode, PKM=PKM at time of error

SETFRR issued in AR ASC mode	ASC mode=AR
	PASN=PASN at the time SETFRR was issued
	SASN=SASN at the time SETFRR was issued
	PKM=PKM at the time SETFRR was issued
		ASC mode=AR
		PASN=PASN at the time SETFRR was issued
		SASN=PASN at the time SETFRR was issued
		PKM=PKM at the time SETFRR was issued
			ASC mode=AR
			PASN=HASN=SASN
			PKM depends on the dispatchable unit mode, as follows:
			• For task mode, see Note 1.
			• For SRB mode, PKM=PKM at time of error

Note:

1. When your program specifies MODE=HOME on SETFRR, and is also in task mode, the PKM for the FRR is the PKM that can be constructed from the storage protection key in the TCB of the currently active task, unless a DAT error occurs. If a DAT error occurs, the PKM is the PKM at the time of the error.
2. The EAX is the same as when the SETFRR macro was issued.
3. The AX is the AX for the current PASN of the recovery routine.
4. On entry to an FRR, the current linkage stack entry is the same as it was at the time of the error, unless a previous FRR added entries to the linkage stack through a PC or BAKR instruction and did not remove them. In such a case, when percolation occurs and the FRR gets control, the linkage stack contains additional entries beyond what was the current entry at the time of the error for which the FRR received control.
5. There is no restriction on AMODE at time of invocation for SETFRR. At time of entry to the recovery routine, the AMODE will be the same as the time of invocation.
6. The PSW key on entry to an FRR is always 0.

Restricted environments: Specifying LOCAL for the MODE parameter on SETFRR indicates to the system that the FRR can run in either a normal environment or a restricted environment. When an FRR is entered in a restricted environment, it is often called a resource manager because its only purpose is to recover resources. These resources are critical address space resources. Requesting to be able to run in a restricted environment is advantageous when the address space under which the unit of work was running terminates. When you specify MODE=LOCAL, the FRR can run in a different address space, and thus can have the opportunity to clean up the critical resource. The system does not allow FRRs entered in a restricted environment to retry.

MODE=LOCAL should be used by services that need to clean up critical address space related resources serialized by means of a local lock (including the CML lock). When LOCAL is specified and the system cannot enter the FRR in its normal mode, it enters the FRR in the restricted LOCAL mode as long as a local lock is held, and the address space whose lock is held has not terminated or suffered a DAT error.

If it is possible for the FRR to get control in one address space in normal mode and in another address space in restricted mode, the FRR must reside in commonly addressable storage.

The system enters an FRR defined with the LOCAL parameter in LOCAL restricted mode for two different reasons:

1. The system tried to establish the environment required to enter the FRR in normal mode but could not; this problem can occur, for example, when the SASN is no longer valid.
2. An address space is terminating and at least one unit of work in that address space is holding the local lock for another address space (CML lock).

   In both cases, the entry environment is:

   - Primary mode, the home address space can be any address space, and the PASN and SASN are the same as the ASN of the locked address space.
   - The system turns on the SDWALCL bit to indicate that the system is entering the routine in LOCAL restricted mode (otherwise, this bit is turned off).
   - The system turns on the SDWACLUP bit to indicate that the routine is not allowed to retry, although, if system conditions permit, nested FRRs are permitted to retry. See "Providing recovery for recovery routines" on page 401 for an explanation of nested recovery routines.

The following considerations apply when the system enters the FRR in LOCAL restricted mode as a result of reason 2:

- If the FRR issues a SETRP macro to request that the system free the CML lock, subsequent FRRs are not entered because the resources in the address space are no longer serialized and therefore no further LOCAL resource clean up can be done.
- The FRR must not depend on running in task mode because, even though the SETFRR macro was issued in task mode, an FRR entered in LOCAL restricted mode runs in SRB mode. In this case, the information in the SDWA reflects the interrupted process that originally held the CML lock, and the FRR is not permitted to retry.
- The FRR created with the LOCAL parameter must be prepared to be suspended during its normal mode recovery processing and then be entered a second time in LOCAL restricted mode to recover critical address space resources. If the FRR that is running has defined another FRR, specifying MODE=LOCAL, the newer FRR gets control in LOCAL restricted mode followed by the FRR that was in control at the time of suspension.

Environment on entry to a retry routine from an FRR

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

Authorization:

Environmental factor	Requirement
Problem or supervisor state	The retry routine from an FRR is always entered in supervisor state.
PSW key	The retry routine from an FRR is always entered in PSW key 0.
PKM	The PKM for a retry routine from an FRR is the same as the PKM for the FRR.

SDWA storage key: For retry from an FRR, the retry routine cannot access the SDWA. The system always frees the SDWA before giving control to the retry routine.

Dispatchable unit mode: The dispatchable unit mode for the retry routine is the mode that existed at the time of error.

Cross memory mode: The PASN and SASN for the retry routine can be either those that existed at the time of the error, or those that existed at the time of the entry to the FRR that is requesting the retry. The FRR makes the choice through the RETRY parameter on the SETRP macro.

When the recovery routine specifies the RETRY=ERROR parameter on SETRP, the system sets the PASN and the SASN to the values they had at the time of the error.

Use caution in choosing to establish the environment that existed at the time of the error, because a problem with the environment might very well be the cause of the error.

You can use the RETRY=FRR parameter on SETRP (or take the default) to specify that the retry routine get control in the cross memory conditions that existed at the time the program issued the SETFRR. Table 62 shows the cross memory conditions that exist if you issue this request.

Note: Specifying MODE=PRIMARY on SETFRR causes the retry routine to lose addressability to what was the secondary address space at the time the macro was issued (the SASN is set to whatever the PASN was at the time SETFRR was issued).

Table 62. Retry to SETFRR Environment (Specifying SETRP RETRY=FRR)

ASC Mode	MODE=FULLXM	MODE=PRIMARY	MODE=HOME

SETFRR issued in primary mode	ASC mode=primary
	PASN=PASN at the time SETFRR was issued
	SASN=SASN at the time SETFRR was issued
		ASC mode=primary
		PASN=PASN at the time SETFRR was issued
		SASN=PASN at the time SETFRR was issued
			ASC mode=primary
			PASN=HASN=SASN

SETFRR issued in secondary mode	ASC mode=secondary
	PASN=PASN at the time SETFRR was issued
	SASN=SASN at the time SETFRR was issued
		ASC mode=primary
		PASN=PASN at the time SETFRR was issued
		SASN=PASN at the time SETFRR was issued
			ASC mode=primary
			PASN=HASN=SASN

Table 62. Retry to SETFRR Environment (Specifying SETRP RETRY=FRR) (continued)

ASC Mode	MODE=FULLXM	MODE=PRIMARY	MODE=HOME

SETFRR issued in AR mode	ASC mode=AR
	PASN=PASN at the time SETFRR was issued
	SASN=SASN at the time SETFRR was issued
		ASC mode=AR
		PASN=PASN at the time SETFRR was issued
		SASN=PASN at the time SETFRR was issued
			ASC mode=AR
			PASN=HASN=SASN

AMODE: Retry routines are entered in the same addressing mode that existed when the recovery routine was entered, unless the SETRP RETRYAMODE= is used by the recovery routine. See the description of RETRYAMODE= parameter on the SETRP information of the *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

ASC mode: The ASC mode on entry to the retry routine depends on what the FRR specifies on the RETRY parameter on SETRP. If the FRR specifies RETRY=ERROR, the system sets the ASC mode to whatever it was at the time of error. If the FRR specifies RETRY=FRR, the ASC mode for the retry routine depends on the ASC mode of the program issuing SETFRR, and on how the program specifies the MODE parameter on SETFRR. See Table 62 on page 386 for the ASC mode values when RETRY=FRR.

Interrupt status: The retry routine is entered disabled if the FRR was entered disabled and does not issue SETRP with FRELOCK=CPU and FRLKRTY=YES. Otherwise, the retry routine is entered enabled.

DU-AL: The retry routine receives control with the same DU-AL that the FRR received, as modified by the FRR. The system does not modify the contents of the DU-AL during recovery processing.

Locks: The status of locks held is the same on entry to the retry routine as it was on entry to the FRR requesting retry. Exceptions occur when the FRR specifies FRELOCK and FRLKRTY on the SETRP macro to request that the system free the locks, or the FRR itself frees the locks. Any locks requested to be freed are no longer held upon entry to the retry routine. If the FRR obtained any locks, the system frees them before giving control to the retry routine.

Program mask: The retry routine is entered with a program mask of zero.

Condition of the linkage stack: The retry routine is always entered with the same linkage stack that was in effect when SETFRR was issued, unless the recovery routine sets the SDWALSLV field.

AX: The AX is the current one for the PASN of the retry routine.

EAX: The EAX is the same as when the SETFRR macro was issued.

Linkage stack at time of retry

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

```
caller ---> PGM1
            BAKR
            :
            ESTAEX
            :
            BALR -------->  PGM2
                            BAKR
                            :
                            BALR -------->  PGM3
                                            BAKR
                                         ***abend***
                                            :
                                            retry point
                                            :
                                 <--------  PR
```

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

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

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

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

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

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

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

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

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

- The system ignores the value when the recovery routine is a STAE, STAI, or FESTAE routine.
- When retry is from an FRR, the value can reflect entries caused by a PC instruction. All other times, the value can reflect only entries caused by a BAKR.
- For unauthorized ESTAE-type recovery routines, all entries must have the same state (problem or supervisor) and have the same PSW key as the program that activated the recovery routine. For authorized ESTAE-type recovery routines or FRRs, this restriction does not apply.
- For ESTAE-type routines, the value must reflect only those entries associated with programs that have been established by a program running under the RB of the retry routine. See “RB considerations” on page 357.

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

Understanding recovery through a coded example

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

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

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

```
EXAMPLE   CSECT                 * SAMPLE PROGRAM THAT USES ESTAEX
EXAMPLE   AMODE 31
EXAMPLE   RMODE ANY
          USING EXAMPLE,15      * ESTABLISH TEMPORARY ADDRESSABILITY
          B    @PROLOG          * BRANCH AROUND EYE CATCHER
          DC   CL24'EXAMPLE 04/10/92.01' * EYE CATCHER
*
* USE THE LINKAGE STACK TO SAVE STATUS ON ENTRY TO THE PROGRAM.
*
@PROLOG   BAKR  14,0            * SAVE REGISTER/PSW STATUS
*
* ESTABLISH ADDRESSABILITY FOR THIS PROGRAM.
*
          LR   12,15            * REG 12 BECOMES BASE REGISTER
          DROP  15              *
          USING EXAMPLE,12      * ESTABLISH ADDRESSABILITY
*
* OBTAIN DYNAMIC STORAGE AREA FOR THIS REENTRANT PROGRAM.
*
          L    2,DYNSIZE        * LENGTH TO OBTAIN
          STORAGE  OBTAIN,ADDR=(1),SP=0,LENGTH=(2)
          LR   13,1             * SAVE DYNAMIC AREA ADDRESS
          USING  DYNAREA,13     * ADDRESSABILITY TO DYNAMIC AREA
*
* SET UP THE REMOTE PARAMETER LIST FOR THE ESTAEX MACRO.
*
          MVC  RMTESTAEX(@LSTSIZE),LSTESTAEX
*
* DEFINE AND ACTIVATE AN ESTAEX RECOVERY ROUTINE AT LABEL 'RECOVERY'.
*
```

```
          ESTAEX RECOVERY,PARAM=DYNAREA,RECORD=YES,MF=(E,RMTESTAEX)
***********************************************************************
*
*  CODE FOR THE MAINLINE ROUTINE FUNCTION CAN BE INSERTED HERE
*
*     IF AN ERROR OCCURS IN THE MAINLINE ROUTINE, THEN THE SYSTEM WILL
*     PASS CONTROL TO RECOVERY.
*
***********************************************************************
*
RETRYPT   DS 0H
***********************************************************************
*
*  CODE FOR THE RETRY ROUTINE FUNCTION CAN BE INSERTED HERE
*
***********************************************************************
          ESTAEX 0              * DELETE THE ESTAEX
          LR   1,13             * FREE DYNAMIC AREA, ADDRESS TO FREE
          L    2,DYNSIZE        * LENGTH TO FREE
          STORAGE  RELEASE,ADDR=(1),SP=0,LENGTH=(2)
          PR                    * RESTORE STATUS & RETURN TO CALLER
***********************************************************************
*
* RECOVERY ROUTINE
*
***********************************************************************
RECOVERY  DS    0H              * ENTRY POINT FOR ESTAEX RECOVERY ROUTINE
*
* HANDLE INPUT FROM THE SYSTEM  AND RE-ESTABLISH ADDRESSABILITY FOR
* BASE REGISTER (12) AND DYNAMIC AREA REGISTER (13)
*
          PUSH  USING
          DROP  ,               * ENSURE NO SPURIOUS USING REFERENCES
          USING RECOVERY,15     * TEMPORARY ADDRESSABILITY
          L     12,#BASE        * RELOAD THE BASE REGISTER
          DROP 15               * RELEASE TEMPORARY ADDRESSABILITY
          USING EXAMPLE,12      * USE THE BASE REGISTER
          USING DYNAREA,13      * DYNAMIC AREA ADDRESSABILITY
          C    0,TESTNOSDWA     * IS THERE AN SDWA PRESENT?
          BE   NOSDWA           * NO, DO NOT USE THE SDWA
HAVESDWA  DS   0H
          USING SDWA,1          * ADDRESSABILITY TO SDWA
          L    13,SDWAPARM      * ADDRESS OF PARAMETER ADDRESS
          L    13,0(13)         * PARAMETER ADDRESS (DYNAREA)
          MVC  SAVE_ABCC,SDWAABCC * SAVE THE COMPLETION CODE
          B    RECOV1           * CONTINUE WITH COMMON RECOVERY
NOSDWA    LR   13,2             * PARAMETER ADDRESS (DYNAREA)
          ST   1,SAVE_ABCC      * SAVE THE COMPLETION CODE
          SR   1,1              * NO SDWA IS AVAILABLE, CLEAR REGISTER
*
* COMMON RECOVERY PROCESSING
*
RECOV1    DS   0H               * COMMON RECOVERY PROCESSING
          ST   1,SAVE_SDWA      * SAVE THE SDWA ADDRESS
          ST   14,SAVE_RETURNR14 * RETURN ADDRESS TO THE SYSTEM
*
***********************************************************************
*
*  CODE FOR THE RECOVERY ROUTINE FUNCTION SHOULD BE INSERTED HERE
*
***********************************************************************
*
* IF THERE IS NO SDWA, THEN SET UP FOR PERCOLATION
*
          L    1,SAVE_SDWA      * RESTORE SDWA REGISTER (1)
          LTR  1,1              * IS THERE AN SDWA?
          BZ   NORETRY          * NO, DO NOT ATTEMPT TO RETRY
```

```
*
* CHECK SDWACLUP TO SEE IF RETRY IS ALLOWED
*
          TM   SDWAERRD,SDWACLUP * IS RETRY ALLOWED?
          BNZ  NORETRY          * NO, DO NOT ATTEMPT TO RETRY
*
* SET UP THE RETURN PARAMETERS TO THE SYSTEM.  THE SETRP MACRO UPDATES
* THE SDWA.  NOTE: THE WKAREA PARAMETER DEFAULTS TO REGISTER 1, WHICH
* HAS THE ADDRESS OF THE SDWA.  ALSO NOTE THAT OTHER REGISTERS MIGHT
* NEED TO BE UPDATED TO MEET THE NEEDS OF DIFFERENT PROGRAMS.
*
          ST     12,SDWASR12    * BASE REGISTER 12 FOR RETRY
          ST     13,SDWASR13    * DYNAMIC AREA REGISTER 13 FOR RETRY
          SETRP  RETREGS=YES,RC=4,RETADDR=RETRYPT,FRESDWA=YES
          B    RECOV2           * CONTINUE WITH COMMON RECOVERY
NORETRY   DS 0H                 * BRANCH HERE WHEN NOT GOING TO RETRY
          LA 15,0               * RETURN CODE TO INDICATE PERCOLATION
RECOV2    DS 0H                 * COMPLETE THE RETURN TO THE SYSTEM
          L   14,SAVE_RETURNR14 * SET THE RETURN ADDRESS TO THE SYSTEM
          BR  14                * RETURN TO THE SYSTEM
*
* STATIC STORAGE AREA
*
TESTNOSDWA DC  F'12'            * TEST FOR NO SDWA CONDITION
#BASE     DC   A(EXAMPLE)       * BASE REGISTER VALUE
DYNSIZE   DC   AL4(@DYNSIZE)    * DYNAMIC AREA SIZE
LSTESTAEX ESTAEX RECOVERY,MF=L  * LIST FORM OF ESTAEX PARAMETER LIST
@LSTSIZE  EQU *-LSTESTAEX       * SIZE OF ESTAEX PARAMETER LIST
*
* DYNAMIC AREA STORAGE FOR REENTRANT PROGRAM
*
DYNAREA   DSECT                 * DYNAMIC STORAGE
SAVEAREA  DS   18F              * REGISTER SAVE AREA
SAVE_SDWA DS   F                * SDWA ADDRESS ON ENTRY TO RECOVERY
SAVE_ABCC DS   F                * COMPLETION CODE
SAVE_RETURNR14 DS F             * RETURN ADDR. TO SYSTEM FROM RECOVERY
RMTESTAEX DS   CL(@LSTSIZE)     * REMOTE ESTAEX PARAMETER LIST
STATUS    DS   F                * MAINLINE STATUS INDICATOR
@ENDDYN   DS   0X               * USED TO CALCULATE DYNAMIC AREA SIZE
@DYNSIZE  EQU  ((@ENDDYN-DYNAREA+7)/8)*8  * DYNAMIC AREA SIZE
*
* INCLUDE MACROS
*
          IHASDWA
          END
```

Understanding advanced recovery topics

This topic contains information about the following advanced recovery topics:

- "Providing recovery with minimal processor overhead (FESTAE macro)" on page 392
- "Invoking RTM" on page 392
- "Providing multiple recovery routines" on page 395
  - "Percolation for the same unit of work" on page 399
  - "SRB-to-task percolation" on page 400
- "Providing recovery for recovery routines" on page 401
- "Providing recovery for multitasking programs" on page 402
- "Using resource managers" on page 402

Providing recovery with minimal processor overhead (FESTAE macro)

A recovery routine defined by the FESTAE macro gets control in the same sequence and under the same conditions as a recovery routine defined by the ESTAE or ESTAEX macro. However, only a type 2, 3, or 4 SVC routine (one for which the system creates an SVRB) can issue FESTAE.

When the SVC routine issues FESTAE, it defines an ESTAE-type recovery routine with minimal processor overhead. The SVC routine can use FESTAE only once to create a recovery routine. Therefore, any SVC routine needing to change its recovery routine address must use branch entry ESTAE services, and any SVC routine needing more than a single recovery routine must use an ESTAEX or ESTAE macro to get the additional recovery routines.

In addition to the parameter area that you can supply by coding the PARAM parameter on the FESTAE macro, a 24-byte parameter area is also available as an option. The name of the optional parameter area is RBFEPARM, which is a field in the RB data area. If you do not specify the PARAM parameter on the FESTAE macro, the system uses RBFEPARM as the default parameter area. When the recovery routine gets control, SDWAPARM (or GPR 2 if the system does not provide an SDWA) contains the address of RBFEPARM. The mainline routine can clear (set to zero) and initialize the parameter area with appropriate information (such as tracking data) that might be useful to the recovery routine. You must clear the parameter area before using it to ensure that no spurious data remains in it from previous processing.

Invoking RTM

RTM can be called to perform its recovery and termination services on behalf of the caller or on behalf of another routine. Two macros -- CALLRTM and ABEND -- invoke RTM.

Using the CALLRTM macro

A routine issues the CALLRTM macro to direct recovery termination services to terminate a task, an address space, or a preemptable SRB. Control returns to the issuer of the macro if TYPE=ABTERM, TYPE=MEMTERM, or TYPE=SRBTERM is specified.

TYPE=ABTERM: If the CALLRTM macro specifies TYPE=ABTERM, system processing is directed toward the specified task. The following locking and save area requirements apply to this form of CALLRTM:

- To terminate the current task, you can specify (or default to) TCB=0, or specify TCB= as the address of the current task's TCB, or specify TTOKEN= as the TTOKEN of the current task. Regardless of how the current task is specified, you must also specify the ASID= parameter using the current ASID.

  In this form of ABTERM, the caller may (but is not required to) hold the local lock. This ABTERM request will be processed asynchronously via an SRB. To prevent the current task from running while the SRB is implementing the requested ABTERM, the current task should be placed into a wait state on an ECB that will never be posted.

  To terminate the current task immediately rather than asynchronously, issue the ABEND macro instead of CALLRTM.
- To terminate a task other than the current task in the same address space, you can either specify TCB= as the address of the target task's TCB, or specify TTOKEN= as the TTOKEN of the target task. Do not specify the ASID=

parameter. In this form of ABTERM, the caller must hold the local lock. To avoid holding the local lock, you can treat the task as one in a different address space and issue the ABTERM with the ASID= parameter.

- To terminate a task in a different address space, you can either specify TCB= as the address of the target task's TCB, or specify TTOKEN= as the TTOKEN of the target task. Regardless of how the target task is specified, you must also specify the ASID= parameter as the target ASID.

  In this form of ABTERM, the caller may (but is not required to) hold the local lock. This ABTERM request will be processed asynchronously via an SRB. It is better to specify the TTOKEN= parameter than the TCB address because the TCB address might be re-used before the SRB actually runs, which could lead to the wrong task being terminated.

Note: The required work area that you provide when you specify either TTOKEN or ASID is not the standard 72-byte save area. The system stores into the first 8 bytes of the area, which are commonly used for save area chaining. If you pass the save area in register 13 that you are using to link your program to your caller's, you will not be able to get back to your caller.

Through the use of the following parameters, valid with TYPE=ABTERM, it is possible to specify what happens once the system receives control.

- DUMP specifies whether a dump should be taken. DUMPOPT and DUMPOPX are optional parameters which allow you to determine the contents of the dump by specifying the address of a parameter list of dump options.
- STEP indicates whether the job step should be abnormally terminated.
- RETRY specifies whether the caller's recovery routines can retry. Specifying RETRY=NO will turn on the SDWACLUP bit and force all recovery routines to percolate rather than retry.

TYPE=MEMTERM: If the CALLRTM macro specifies TYPE=MEMTERM, system processing is directed toward an address space. The following locking and work area requirements apply to this form of CALLRTM:

- If the ASID parameter is nonzero, the specified address space is abnormally terminated. The caller need not be disabled or own any locks. The caller must pass the address of a 72-byte work area in GPR 13.
- If the ASID parameter is specified as 0 or is omitted, the current address space is abnormally terminated. The caller need not be disabled or own any locks. The caller must pass the address of a 72-byte work area in GPR 13.

  Note: The required work area is not the standard 72-byte save area. The system stores into the first 8 bytes of the area, which are commonly used for save area chaining. If you pass the save area in register 13 that you are using to link your program to your caller's, you will not be able to get back to your caller.

Because TYPE=MEMTERM processing circumvents all task recovery and task resource manager processing, its use is restricted to a select group of routines that can determine that task recovery and task resource manager clean-up are either not warranted or will not successfully operate in the address space being terminated. An alternative way to terminate an address space is to use CALLRTM TYPE=ABTERM and specify the job step TCB.

TYPE=SRBTERM: If the CALLRTM macro specifies TYPE=SRBTERM, system processing is directed toward a specific preemptable SRB, identified by the

SRBIDTOKEN provided via IEAMSCHD. The following locking and work area requirements apply to this form of CALLRTM:

- The caller may hold a lock, but none are required.
- TYPE=SRBTERM requires a 144-byte workarea that is not used as a standard savearea (like the 72-byte workarea required by TYPE=ABTERM).
- An ASID should not be specified with TYPE=SRBTERM. The SRBIDTOKEN provides the necessary information for the system to find the SRB.
- The target SRB is processed asynchronously and may terminate after control has returned to the invoking program.
- RETRY specifies whether the caller's recovery routines can retry. Specifying RETRY=NO will turn on the SDWACLUP bit and force all recovery routines to percolate rather than retry.
- Preemptable SRBs can be protected from CALLRTM TYPE=SRBTERM,SRBIDTOKEN=*token addr* by holding a lock or using the process-must-complete mode.

For TYPE=SRBTERM, set the high-order bit of the reason code to 1 when you want to indicate that an SVCDUMP is not necessary for the abend being issued. The system does not do anything special with this bit, but recovery routines can use the information when determining if they should request an SVCDUMP of the abend.

Using the ABEND macro

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

The position within the job step hierarchy of the task for which the ABEND macro is issued determines the exact function of the abnormal termination routine. If an ABEND macro is issued when the job step task (the highest level or only task) is active, or if the STEP parameter is coded in an ABEND macro issued during the performance of any task in the job step, all the tasks in the job step are terminated. For example, if the STEP parameter is coded in an ABEND macro under TSO/E, the TSO/E job is terminated. An ABEND macro (without a STEP parameter) that is issued in performance of any task in the job step task usually causes only that task and its subtasks to be abnormally terminated. However, if the abnormal termination cannot be fulfilled as requested, it might be necessary for the system to end the job step task abnormally.

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

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

- Releases the resources owned by the terminating task and all of its subtasks, starting with the lowest level task.
- Places the system or user completion code specified in the ABEND macro in the task control block (TCB) of the active task (the task for which the ABEND macro was issued).

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

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

- Releases the resources owned by each task, starting with the lowest level task, for all tasks in the job step. The system does not give control to any end-of-task exit.
- Writes the system or user completion code specified in the ABEND macro on the system output device.

The remaining steps in the job are skipped unless you can define your own recovery routine to perform similar functions and any other functions that your program requires. Use either the ESTAE or ESTAEX macro, or the ATTACHX macro with the ESTAI option to provide a recovery routine that gets control whenever your program issues an ABEND macro. If your program is running in AR ASC mode, use the ESTAEX or ATTACHX macro.

Providing multiple recovery routines

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

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

MVS functions provide their own recovery routines; thus, percolation can pass control to both installation-written and system-provided recovery routines. If all recovery routines percolate -- that is, no recovery routine can recover from the error -- then the unit of work (an SRB or a task) is terminated.

When a recovery routine gets control and cannot recover from the error (that is, it does not retry), it must free the resources held by the mainline routine and request that the system continue with error processing (percolate). Note that a recovery routine entered with the SDWACLUP bit set to one, indicating that retry is not permitted, has no choice but to percolate. When the recovery routine requests percolation, the previously activated recovery routine gets control. When a retry is not requested and the system has entered all possible recovery routines, the unit of work (either an SRB or a task) ends abnormally. Figure 59 on page 397 shows the decisions the system makes to determine which recovery routine is to get control in a particular situation.

When a recovery routine requests percolation, it is deactivated; that is, it can no longer get control for this error. A deactivated recovery routine is not entered again unless that recovery routine is activated again after a retry. There are two types of percolation: percolation for the same unit of work and SRB-to-task percolation.

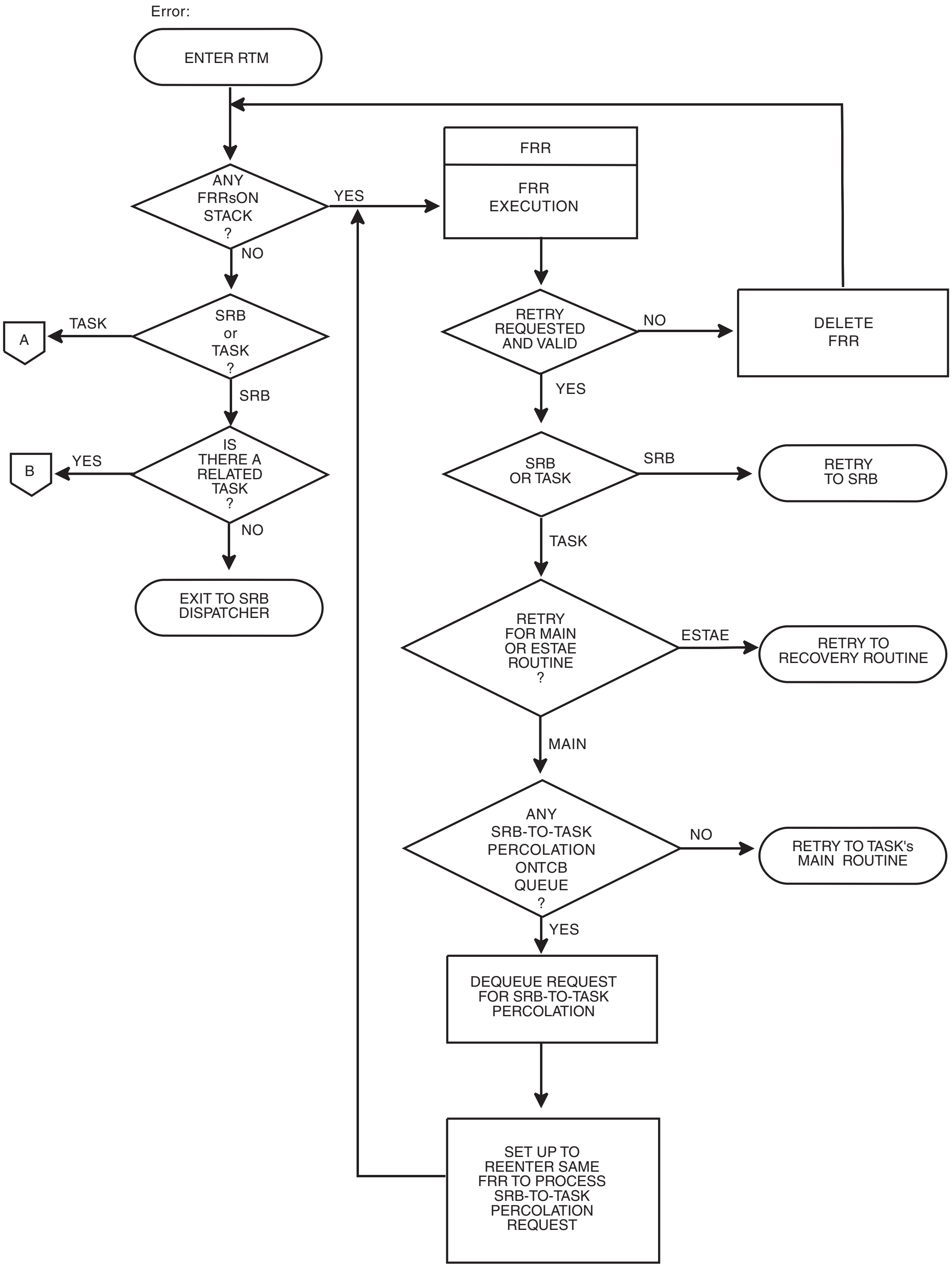

Error:
ENTER RTM
ANY FRRsON STACK ?
YES
NO
FRR
FRR EXECUTION
RETRY REQUESTED AND VALID
NO
DELETE FRR
YES
SRB or TASK ?
TASK
A
SRB
IS THERE A RELATED TASK ?
YES
B
NO
EXIT TO SRB DISPATCHER
SRB OR TASK
SRB
RETRY TO SRB
TASK
RETRY FOR MAIN OR ESTAE ROUTINE ?
ESTAE
RETRY TO RECOVERY ROUTINE
MAIN
ANY SRB-TO-TASK PERCOLATION ONTCB QUEUE ?
NO
RETRY TO TASK's MAIN ROUTINE
YES
DEQUEUE REQUEST FOR SRB-TO-TASK PERCOLATION
SET UP TO REENTER SAME FRR TO PROCESS SRB-TO-TASK PERCOLATION REQUEST

Figure 59. Routing control to recovery routines, example 1

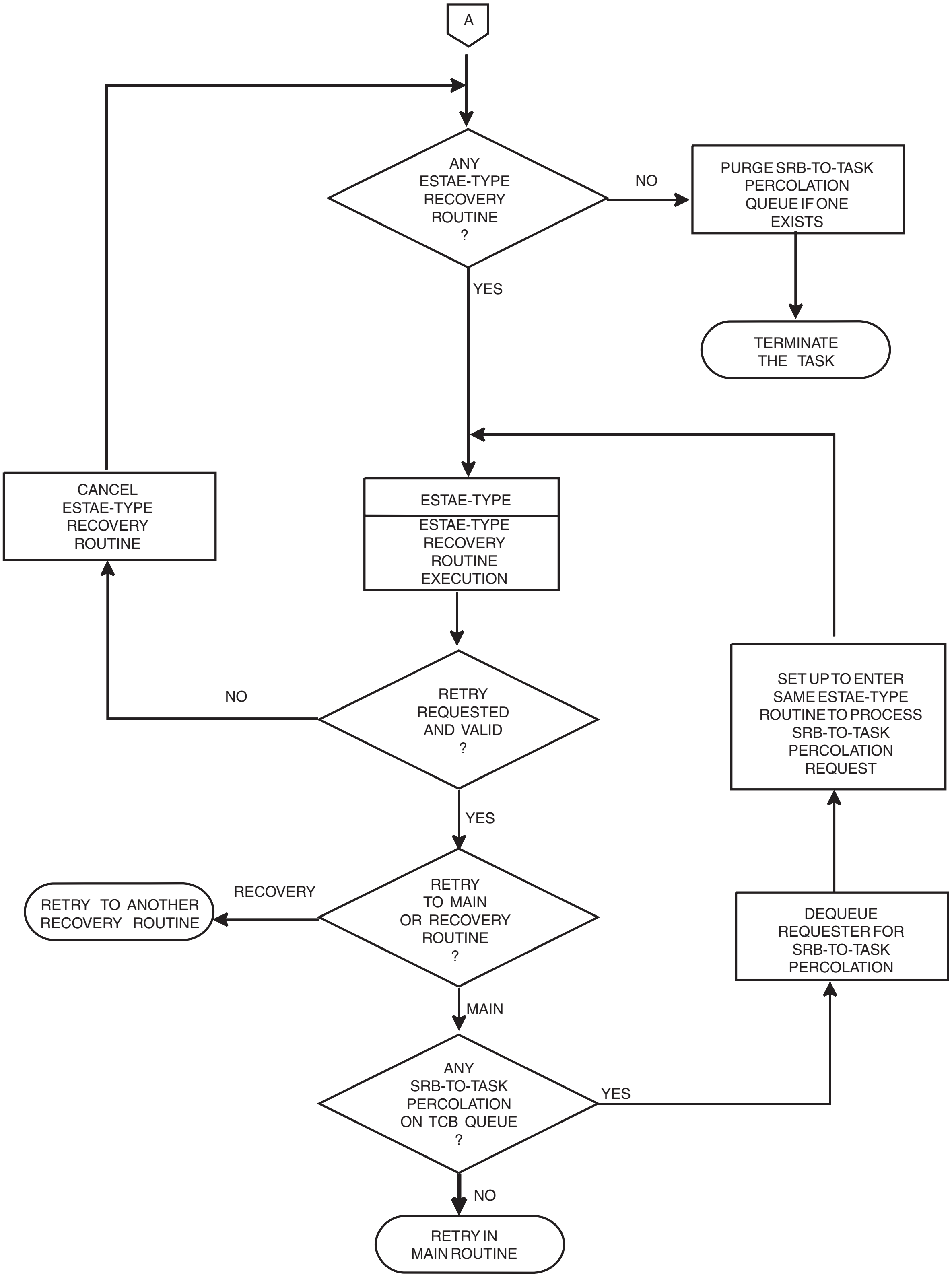

A
ANY ESTAE-TYPE RECOVERY ROUTINE ?
NO
PURGE SRB-TO-TASK PERCOLATION QUEUE IF ONE EXISTS
TERMINATE THE TASK
YES
CANCEL ESTAE-TYPE RECOVERY ROUTINE
ESTAE-TYPE
ESTAE-TYPE RECOVERY ROUTINE EXECUTION
RETRY REQUESTED AND VALID ?
NO
YES
SET UP TO ENTER SAME ESTAE-TYPE ROUTINE TO PROCESS SRB-TO-TASK PERCOLATION REQUEST
RETRY TO MAIN OR RECOVERY ROUTINE ?
RECOVERY
RETRY TO ANOTHER RECOVERY ROUTINE
DEQUEUE REQUESTER FOR SRB-TO-TASK PERCOLATION
MAIN
ANY SRB-TO-TASK PERCOLATION ON TCB QUEUE ?
YES
NO
RETRY IN MAIN ROUTINE

Figure 60. Routing control to recovery routines, example 2

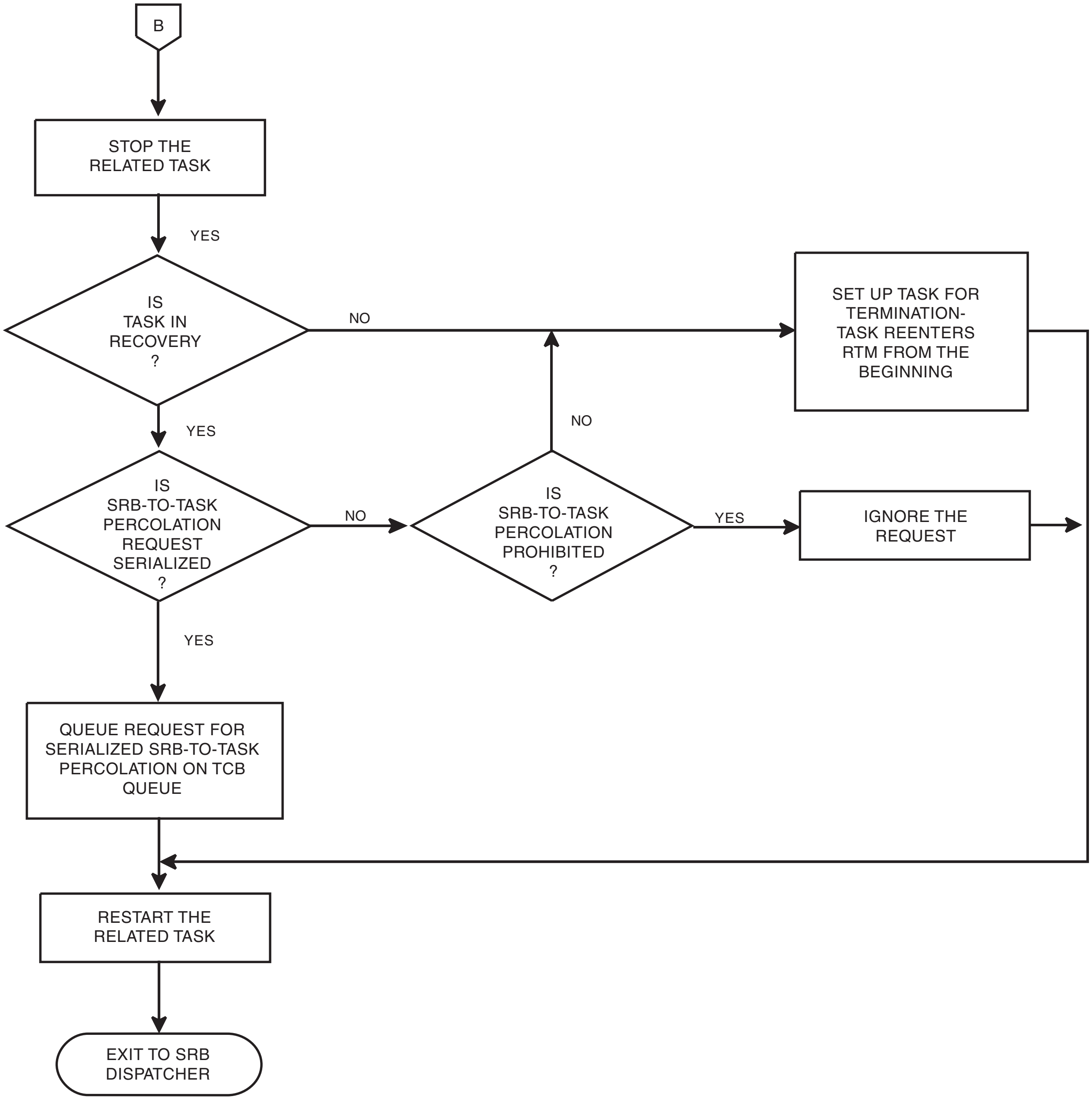

B
STOP THE RELATED TASK
YES
IS TASK IN RECOVERY ?
NO
SET UP TASK FOR TERMINATION-TASK REENTERS RTM FROM THE BEGINNING
YES
NO
IS SRB-TO-TASK PERCOLATION REQUEST SERIALIZED ?
NO
IS SRB-TO-TASK PERCOLATION PROHIBITED ?
YES
IGNORE THE REQUEST
YES
QUEUE REQUEST FOR SERIALIZED SRB-TO-TASK PERCOLATION ON TCB QUEUE
RESTART THE RELATED TASK
EXIT TO SRB DISPATCHER

Figure 61. Routing control to recovery routines, example 3

Percolation for the same unit of work

Percolation for the same unit of work causes control to be given to one recovery routine after another for that same unit of work, which can be either a task or an SRB.

Percolation to an FRR always occurs from another FRR. If an FRR obtains locks that were not held when it was entered and then requests percolation, the system frees those locks before giving control to the next FRR. Also, if the percolating FRR requested that the system free any locks, the system frees these locks before giving control to the next FRR.

Percolation to an ESTAE-type recovery routine can occur from either an FRR or another ESTAE-type recovery routine. The environment in which an ESTAE-type recovery routine gets control does not vary regardless of whether the percolation request came from an FRR or another ESTAE-type recovery routine. Note that a recovery routine defined by the ESTAI parameter of the ATTACHX macro can choose either to percolate to a previous ESTAI routine (by setting a return code of 0 for the system) or to bypass further ESTAI recovery routine processing and continue with error processing (by setting a return code of 16 for the system).

SRB-to-task percolation

When an SRB is scheduled and the fields SRBPASID and SRBPTCB are supplied, or IEAMSCHD is used with the PTCBADDR= parameter and SYNCH=NO, the specified TCB is defined as the SRB's related task. This TCB is in the ASID specified by SRBPASID or the Home ASID of the issuer of IEAMSCHD. When an SRB with a related task ends abnormally and the FRR for the SRB does not exist or does not request a retry, the error is percolated to the recovery routine for the related task. This percolation is called SRB-to-task percolation.

SRB-to-task percolation occurs if none of the FRRs defined by the SRB retry, or if the SRB does not have an FRR. Either case creates a request for the system to perform SRB-to-task percolation. The system ignores the request whenever the related task has already ended. If serialization is requested on the SETRP macro in the FRR for that SRB, the percolation request is deferred if the task is already in recovery. (See the SERIAL=YES parameter of the SETRP macro in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.) Serializing SRB-to-task percolation ensures that information about multiple SRB failures is not lost.

Note: SERIAL=YES should not be specified unless the task's recovery routine expects it.

The system processes requests for non-serialized SRB-to-task percolation as follows: the system abnormally ends the task and passes the information about the SRB's error to the most recently activated recovery routine for the task.

The system processes requests for serialized SRB-to-task percolation as follows:

- If the task is already in recovery, the system saves and queues the information about the SRB's error for processing later when the task recovers from the previous error.
- If the task is not in recovery, the system abnormally ends the task and passes the information about the SRB's error to the task's recovery.

When one of the task's recovery routines that is not a nested recovery routine (defined in “Providing recovery for recovery routines” on page 401) requests a retry and retry is allowed, the system checks for queued requests for SRB-to-task percolation and takes the following actions before performing the retry:

1. If any requests are queued, the system dequeues a request and again enters the recovery routine that requested the retry. The system repeats this process as long as there are queued requests.
2. When the queue is empty or depleted, the system honors the retry request and gives control to the retry routine.

Figure 59 on page 397 shows this process.

The environment for a task recovery routine entered as a result of SRB-to-task percolation is the same as the environment described earlier under “Percolation for the same unit of work” on page 399.

However, the information in the SDWA describes the error that occurred in the SRB. Whenever serialized SRB-to-task percolation is requested and the system must queue a request, the system obtains an area from the user's private area to preserve the information about the SRB's error. If no space is available, the system cannot preserve that information but still enters the task recovery for the request. If an SDWA is available, the system sets the SDWARPIV bit to indicate that error-time information is not available. Also, the system sets the SDWACOMU field to zeros because the system cannot preserve its contents to pass from the SRB's FRR to the task's recovery routine.

Providing recovery for recovery routines

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

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

You can activate an FRR from either another FRR or from an ESTAE-type recovery routine. When activating an FRR from an ESTAE-type recovery routine, the ESTAE-type recovery routine must have acquired a lock after being entered, or the FRR must be an EUT FRR.

You should activate an ESTAE-type recovery routine only from an ESTAE-type recovery routine. **IBM recommends you do not** activate an ESTAE-type recovery routine from an FRR, because it might not get control in the correct order.

Any recovery routine activated from a recovery routine is called a nested recovery routine. A nested recovery routine can retry provided the error for which it gets control is retryable. Whether the nested recovery routine can retry is independent of whether the recovery routine that activated the nested recovery routine can retry. For example, a recovery routine might get control for a non-retryable error. The recovery routine itself might encounter an error for which its recovery routine (the nested recovery routine) gets control and can retry. The retry routine from a nested ESTAE or ESTAEX recovery routine runs under the RB of the ESTAE-type recovery routine that activated the nested recovery routine.

Nested FRRs do not have access to their owning FRR's SDWA or 304-byte work area. The system makes a temporary copy of these areas and re-uses the original SDWA and work area when giving control to the nested FRR. Nested FRRs do have access to their owning FRR's 24-byte parameter area.

Providing recovery for multitasking programs

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

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

One serialization technique to ensure the order of termination processing is to use the DETACH macro. Issuing DETACH ensures that the detached task and its recovery routines complete before processing for the issuing task proceeds. DETACH can only be used for tasks that were directly attached by the recovery routine's TCB.

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

Using resource managers

MVS provides resource managers that are invoked to “clean up” resources associated with a task or an address space. A resource manager is a routine that gets control during normal and abnormal termination of a task or an address space. Task or address space termination is the process of removing a task or address space from the system, releasing the resources from the task or address space, and making the resources available for reuse.

During normal termination, RTM invalidates any existing recovery except for STAI/ESTAI recovery routines. 'End of task' resource managers are called, then the task ends. If an ABEND occurs while an 'end of task' resource manager is running, RTM will call any recovery that was established by the resource manager. If that recovery percolates (or is not allowed to retry), then RTM will give control to eligible STAI/ESTAI exits. These STAI/ESTAI exits are not allowed to retry. The task will then enter abnormal termination at the point where we begin to call 'end of task' resource managers as described above. Since we start over again at the beginning of resource manager processing, it is possible for a resource manager to run twice -- once for normal termination and once for abnormal termination.

Abnormal termination begins after an ABEND occurs and all recovery exits have run and percolated. After this, RTM detaches any remaining subtasks, then calls 'end of task' resource managers for an abnormal termination. These resource managers are protected from cancels, detaches and 'SRB-to-task' percolation ABENDs. If an ABEND occurs while a resource manager is running, any recovery that it established is given control and may retry if the ABEND is able to be retried. If the recovery does not retry, RTM continues by calling the next resource manager. After all resource managers have run, the task ends.

'End of memory' resource managers run in the master address space (ASID 1) and do not have access to local storage in the terminating address space. If an 'end of memory' resource manager abends, any recovery that it has established will receive control and be allowed to retry if the ABEND is able to be retried. If the recovery does not retry, RTM continues by calling the next resource manager. After all resource managers have run, the address space is terminated.

If an 'end of memory' resource manager remains dormant for more than four minutes, the system will consider the resource manager to have 'hung' and issues ABEND 30D after scheduling an SVCDUMP of the hang. The resource manager's recovery is allowed to retry ABEND 30D. For more details, see the description of the ABEND 30D in *z/OS MVS System Codes*.

Resource manager execution environment

Task-related resource managers will receive control in different environments depending on the parameters passed into the RESMGR macro, but they all have some common features. All resource managers will receive control in task mode, enabled for I/O and external interrupts, and with no locks held.

Control parameters are also provided to all routines and include the RMPL, a user-supplied PARAM value, and the resource manager work area. These areas are in key 0 storage (below the 16M line) in the home address space.

All other environmental settings depend on how the resource manager was originally set up, whether it is for branch, link or PC entry.

Environment for ROUTINE=(BRANCH,routine):

Environmental factor	Requirement
State and key:	Supervisor state, Key 0
Cross memory mode:	Home = Primary = Secondary
AMODE:	AMODE 31
ASC mode:	Primary

Environment for ROUTINE=(LINK,routine): The routine must reside within the Link Pack Area (LPA) or an APF-authorized library in the LNKLST set that is active when the LINK is issued.

Environmental factor	Requirement
State and key:	Supervisor state, Key 0
Cross memory mode:	Home = Primary = Secondary
AMODE:	The AMODE of the target program (24 or 31, AMODE of 64 is not supported).
ASC mode:	Primary

Environment for ROUTINE=(PC,routine): A PC routine must be defined so that it may receive control from supervisor state, key 0.

Environmental factor	Requirement
State and key:	State and Key as defined via the ETDEF for the PC
Cross memory mode:	If it is not a space switcher PC, then Home = Primary = Secondary. Otherwise, the ETDEF of the PC will define the cross memory mode.
AMODE:	Defined via the ETDEF for the PC
ASC mode:	ASC mode defined via ETDEF for the PC

Installation-written resource managers

If an installation creates a function that uses a resource that cannot be protected by an ESTAE-type recovery routine or an FRR, the installation should write a resource manager for that function. The installation-written resource manager should perform the same basic tasks for this new function that system resource managers perform for system functions. These are:

- **At task termination**: ensure that the terminating task is not connected to or associated with any resources; that is, each resource (data set, volume, device) is left in such a state that another task in the address space or system can reuse it.
- **At address space termination**: release all system queue area and common service area control blocks obtained for the use of the terminating address space. Reset all buffers, bit settings, pointers, and so on relating to the address space.

In addition, the installation-written resource manager can include any special processing the installation requires, and can provide recovery routines to protect its function.

Adding an installation-written resource manager

An installation can define an installation-written resource manager either dynamically (while the system is running) or at IPL.

Note

IBM recommends that you use the RESMGR macro to define resource managers dynamically rather than using CSECT IEAVTRML at IPL.

The RESMGR macro:

- Does not require an IPL, and
- Can specify a resource manager for specific address spaces or tasks and can specify the type of linkage, which can improve the performance of your system.

Dynamically: By using the RESMGR macro, an authorized routine can dynamically add (or delete) a resource manager. If you want to provide a routine that disposes of your address space and task resources in a particular way, use RESMGR to add the routine to the system. When you use RESMGR, the routine that you add takes effect immediately and before the next IPL of the system. Use RESMGR to delete the routine when it is no longer required.

The resource manager that you add can be one that monitors the termination of one or more address spaces, or one or more tasks. The following rules apply before you can issue RESMGR to provide a resource manager:

- To monitor a single task in the home address space, that task must exist.

  Note: To monitor for several tasks, issue RESMGR for each task.
- To monitor all tasks in the home address space, that address space must exist, but a particular task need not exist.
- To monitor all tasks in all address spaces, a particular address space or task need not exist.
- To monitor a single address space, that address space must exist.
- To monitor all address spaces, a particular address space need not exist.

The system automatically deletes the resource manager if it is monitoring the termination of the following:

- A specific task, when the task terminates
- A specific address space, when the address space terminates
- All tasks in a specific address space, when the address space terminates.

The system does not automatically delete the resource manager if it is monitoring termination of the following (the resource manager stays in the system to monitor for the next termination):

- All tasks in all address spaces
- All address spaces.

Whether or not the system is to automatically delete the resource manager, you can invoke the delete service of RESMGR at any time to force the deletion of the resource manager.

For a description of the RESMGR macro, see *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

At IPL: To add your own resource managers for installation applications at IPL, place their names in the CSECT IEAVTRML, which is provided by MVS.

Initially, IEAVTRML consists of four 12-byte entries, each containing zeros. You can modify each of the first three 12-byte entries (using the AMASPZAP service aid as described in *z/OS MVS Diagnosis: Tools and Service Aids*) to contain a module name in the first eight bytes; the last four bytes of each entry are reserved and always contain zeros. The last entry must also contain all zeros, to indicate the end of the list. A typical entry for the CSECT might be:

```
DC   CL8'MODULENM'
DC   XL4'00'
```

To add the names of more than three installation-written resource managers, create an entry for each module and a final entry that contains all zeros. Then assemble your modified IEAVTRML and use the modified CSECT to replace the existing IEAVTRML module in load module IGC0001C in SYS1.LPALIB. Place each installation-written resource manager in SYS1.LINKLIB (or a library concatenated to SYS1.LINKLIB through a LNKLSTxx member of SYS1.PARMLIB) or SYS1.LPALIB. If every routine named in IEAVTRML is not present in one of these libraries, the IPL fails.

Using the IEAVTRML method to add installation-written resource managers is still supported, but is no longer recommended because of performance considerations. IEAVTRML exits receive control by means of a LINK that will be issued when both EVERY task in EVERY address space terminates and EVERY address space terminates. Placing an exit routine in the LNKLST concatenation has caused significant performance delays during concurrent IPLs of multiple systems in a sysplex. In extreme cases, task termination delays can result in storage shortage in the address space that delays and might prevent successful IPL.

The resource manager parameter list

The resource manager parameter list (RMPL) is the interface between MVS and the installation-written resource manager. The RMPL tells the resource manager why it was invoked and provides information for its use during processing. RMPL fields indicate, for example, whether the resource manager is being invoked during task termination or address space termination, and whether the termination is normal or abnormal.

To access the contents of the RMPL, the resource manager must include the IHARMPL mapping macro, which provides the field names and describes their content and use. Detailed information on the name, offset, and meaning of each field in the RMPL appears in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Register use

On entry to the resource manager, register contents are:

Register
Contents

0 Does not contain any information for use by the routine.

1 Address of an 8-byte field. The first 4 bytes of this 8-byte field contain the address of the resource manager parameter list (defined in your routine by the IHARMPL mapping macro).

If this is a dynamically invoked resource manager, the second 4 bytes contain the address of an 8-byte field that contains a user-supplied value on the PARAM parameter of the RESMGR macro. If the resource manager was not dynamically added, this 4-byte field contains a zero.

2-12 Do not contain any information for use by the routine.

13 Address of a standard 72-byte save area.

14 Return address.

15 Entry point address in the resource manager.

Your resource manager must save and restore registers 1-14, and may use registers 15 and 0 respectively to pass a return and reason code to the system. When the resource manager is not established with the RESMGR macro, all return and reason codes are ignored. When the resource manager is established with the RESMGR macro, all return codes will be ignored except return code 8. When the return code in register 15 is 8, the system will honor the following reason codes in register 0:

1 Indicates that the system should remove the returning resource manager. Usage of this reason code is equivalent to issuing a RESMGR macro with the DELETE keyword and the appropriate parameters for deleting the returning resource manager. Note that the ECB keyword of RESMGR is not provided by this interface.

Processing sequence

The system gives control to the installation-written resource managers in the following order:

1. Resource managers listed in the CSECT IEAVTRML.
2. Dynamic resource managers, either address space or task, get control in LIFO order within resource manager type.

   Dynamic task resource managers are processed in the following order:

   a. Resource managers provided to protect the specific terminating task
   b. Resource managers provided to protect all tasks in the home address space of the terminating task
   c. Resource managers provided to protect all tasks in all address spaces.

   Dynamic address space resource managers are processed in the following order:

   a. Resource managers provided to protect a specific address space
   b. Resource managers provided to protect all address spaces.

For task termination, the resource manager runs under an RB for the terminating task. For address space termination, it runs in task mode in the master scheduler address space. In either case, the resource manager gets control in PSW key 0, supervisor state, with no locks held.

After all installation-written resource managers have completed processing, the system invokes the system-provided resource managers.

Using STAE/STAI routines

Note:

1. **IBM recommends** you use the ESTAEX or ESTAE macro, or the ESTAI parameter on ATTACHX.
2. Under certain circumstances, STAE or STAI routines might receive control in a restricted environment. See “Restricted environments” on page 376 for more information.

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

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

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

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

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

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

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

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

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

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

If an SDWA was obtained:

GPR **Contents**

0 A code indicating the type of I/O processing performed:

- **0** Active I/O has been quiesced and is restorable.
- **4** Active I/O has been halted and is not restorable.
- **8** No active I/O at abend time.
- **16 (X'10')**
  Active I/O, if any, was allowed to continue.

1 Address of the SDWA.

2 Address of the parameter area you specified on the PARAM parameter.

3-12 Do not contain any information for use by the routine.

13 Save area address.

14 Return address.

15 Address of STAE recovery routine.

If no SDWA was available:

GPR **Contents**

0 12 (X'0C') to indicate that no SDWA was obtained.

1 Completion code.

2 Address of user-supplied parameter list.

3-13 Do not contain any information for use by the routine.

14 Return address.

15 Address of STAE recovery routine.

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

Return Code
Action

0 Continue the termination. The next STAI, ESTAI, or ESTAE routine will be given control. No other STAE routines will receive control.

4,8,12 A retry routine is to be scheduled.

Note: These values are not valid for STAI/ESTAI routines that receive control when a resource manager fails during **normal** termination of a task. See "Restricted environments" on page 376 for more information.

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

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

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

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

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

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

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

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

If there are quiesced restorable input/output operations (as specified by PURGE=QUIESCE on the macro invocation), the retry routine can restore them in the same manner as the retry routine from an ESTAE routine. See "Outstanding I/Os at the time of failure" on page 359.

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

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

When the storage is no longer needed, the retry routine should use the FREEMAIN macro to free the first 104 bytes of the SDWA. If the retry routine is in the user key, this storage should be freed from subpool 0 which is the default subpool for the FREEMAIN macro. If the retry routine is in the system key, storage must be freed from subpool 250.

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

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

The retry routine is entered in supervisor state if the RBOPSW of the retry RB is in supervisor state and the task was authorized at the time the STAE routine was activated or at the time of the error. Otherwise, the retry routine is entered in problem state.

The task is considered to be authorized at the time the STAE routine is activated when at least one of the following is true:

- The task is APF-authorized.
- The requestor is in supervisor state.
- The requestor has a PSW key less than 8.
- The task has a protect key less than 8.
- The PKM of the requestor allows keys less than 8.

The mainline routine is considered to be authorized at the time of the error when at least one of the following is true:

- The task is APF-authorized.
- The task in error has a protect key less than 8.
- All RBs for the task in error run in supervisor state.

The retry routine is entered with the same PSW key as the one in RBOPSW of the retry RB when one of the following is true:

- The task was authorized at the time of the error as described above.
- The RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem state, and the PKM of that RB does not have authority to keys less than 8.

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

Chapter 19. Processing program interruptions (SPIE, ESPIE)

The SPIE[1] macro enables a problem program executing in 24-bit addressing mode to specify an error exit routine to get control in response to one or more program error interruptions. The ESPIE macro extends the function of SPIE to callers in 31-bit addressing mode. Callers in both 24-bit and 31-bit addressing mode can use the ESPIE macro.

If a program interruption occurs, the SPIE/ESPIE exit receives control only when all of the following conditions exist:

- The interrupted program is a problem state program.
- The program interrupt code is in the range 1–15 and 17.
- For the SPIE macro, the interrupted program is in primary address space control (ASC) mode. For the ESPIE macro, the interrupted program is in primary or access register (AR) ASC mode. For both the SPIE and ESPIE macros, the interrupt occurred when the primary, home, and secondary address space were the same.

Note: SPIE/ESPIE exits *cannot* percolate to FRR or ESTAE-type recovery routines.

SPIE/ESPIE exits that were established for the specific program interruption that occurred get control before any RTM (FRR or ESTAE-type) routines, so RTM is disabled for this unit of work. However, if the SPIE/ESPIE exit encounters an error, RTM *will* process this secondary error.

A SPIE exit routine established while executing in 24-bit addressing mode does not receive control if the program executing is in 31-bit addressing mode at the time of the interruption.

Each succeeding SPIE/ESPIE macro completely overrides any previous SPIE/ESPIE macro specifications for the task. The specified exit routine gets control in the key of the TCB when one of the specified program interruptions occurs in any problem program of the task. When a SPIE macro is issued from a SPIE exit routine, the program interruption element (PIE) is reset (zeroed). Thus, a SPIE exit routine should save any required PIE data before issuing a SPIE.

If a caller issues an ESPIE macro from within a SPIE exit routine, it has no effect on the contents of the PIE. However, if an ESPIE macro deletes the last SPIE/ESPIE environment, the PIE is freed, and the SPIE exit cannot retry.

If the current SPIE environment is cancelled during SPIE exit routine processing, the system will not return to the interrupted program when the SPIE program terminates. Therefore, if the SPIE exit routine wishes to retry within the interrupted program, a SPIE cancel should not be issued within the SPIE exit routine.

The SPIE macro can be issued by any problem program being executed in the performance of the task. The system automatically deletes the SPIE exit routine when the request block (RB) that created the SPIE macro terminates.

[1] IBM recommends that you use the ESPIE macro rather than SPIE.

When using Floating Point Registers (FPRs) an ESPIE routine will receive the DXC value in its parameter area, and should use this value rather than the value in the Floating Point Control (FPC) register.

If a retry is to be done, an ESPIE routine can manually change the values of the FPRs and FPC register. Changes to the nonvolatile fields (that is, the IEEE settings) in the FPC register must be made carefully as this could affect the processing of the rest of the current program, and possibly subsequent programs.

The SPIE and ESPIE macros and their related services are discussed in detail in *z/OS MVS Programming: Assembler Services Guide*, *z/OS MVS Programming: Assembler Services Reference ABE-HSP*, and *z/OS MVS Programming: Assembler Services Reference IAR-XCT*. The syntax of both the SPIE and the ESPIE macros appears in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

Interruption types

You can specify interruptions 1-15 and 17 using either the SPIE or the ESPIE macro. Interruption 17 designates page faults and can be specified so that a user-written SPIE/ESPIE exit routine gets control before a supervisor routine when a problem state page fault occurs. The user-provided SPIE/ESPIE exit routine gets control in problem program state and in the key of the TCB when a page fault occurs for the program that issued the SPIE/ESPIE macro. The exit routine gets control in the addressing mode that was in effect when the SPIE or ESPIE macro was issued. (If a SPIE macro was issued this is 24-bit addressing mode.) The SPIE/ESPIE exit routine for interruption type 17 handles page faults at the task level. This includes all RBs executing under the task for which the SPIE/ESPIE was issued. The exit routine resolves page faults by invoking the paging supervisor. In order to specify interruption 17, your program must have APF authorization or be executing in system key (0-7).

A caller in supervisor state who issues the SPIE macro is abnormally terminated with a X‘30E’ abend completion code. A caller in supervisor state who issues the ESPIE macro is abnormally terminated with a 46D-18 abend completion-reason code. If the caller takes a page fault while in supervisor state, the exit routine does not get control even if a SPIE/ESPIE macro specifying interruption type 17 is in effect. Supervisor routines resolve the page fault and continue program processing without abending the caller.

If a program fault occurs while a SPIE or ESPIE that specifies interrupt type 17 is in effect, the SPIE/ESPIE exit routine receives control. If a page fault occurs during setup processing while referencing the PIE/PICA (for a SPIE) or the EPIE (for an ESPIE), the system handles the original page fault. In this case, the SPIE/ESPIE exit routine does not handle the original page fault because the system cannot provide the information that the exit routine needs. After the page fault is resolved, processing continues in the problem program.

There is another situation in which the exit routine might not get control when a page fault occurs. To use interruption type 17, you must place the PIE/PICA (for a SPIE macro), the EPIE (for an ESPIE macro), the SPIE/ESPIE program, and data areas in fixed storage. Page faults can occur after issuing the SPIE/ESPIE macro and before placing this information in fixed storage. If a page fault occurs at this point, the SPIE/ESPIE service routine performs set-up processing and, if it can reference the PIE/PICA (for SPIE) and the EPIE (for ESPIE), passes control to the exit routine. If the exit routine encounters a page fault, the paging supervisor resolves the page fault unless the routine is running disabled. A disabled page fault causes an 0C4 abend. Once the page fault is resolved, normal processing continues in the exit routine.

The ESPIE routines for interruption type 17 do *not* get control on data space faults. The ESPIE macro is not designed to be driven for disabled faults, segment or region faults, or cross memory faults (data space faults might be considered as a special case of cross memory faults because the faulted space is not the home or primary address space).

Chapter 20. Dumping virtual storage (SDUMPX, SDUMP, and IEATDUMP macros)

Where To Find Prerequisite Information

ABEND and SNAP dumps are available to both authorized and unauthorized programs. *z/OS MVS Programming: Assembler Services Guide* introduces ABEND and SNAP dumps and discusses functions that are available to all programs. The following topic discusses SVC dumps, which are available to authorized programs only and Transaction dumps which are available to both authorized and unauthorized programs. Before reading the following topic, read about ABEND and SNAP dumps in *z/OS MVS Programming: Assembler Services Guide*.

When an error occurs, a program can request a dump of virtual storage that a programmer can use to diagnose the error. To request a dump of the storage, the program can issue one of the following:

- The ABEND macro with the DUMP parameter, or the SETRP macro with DUMP=YES. These two macros produce an ABEND dump.
- The CALLRTM macro with DUMP=YES. This macro also produces an ABEND dump.
- Either the SDUMPX or SDUMP macro. These macros produce an unformatted SVC dump that is written to a SYS1.DUMPnn data set. **IBM recommends using SDUMPX** over SDUMP, although MVS accepts SDUMP macro invocations.
- The IEATDUMP macro. This macro produces an unformatted Transaction dump that is written to a pre-allocated or automatically allocated data set.

SVC dumps

This information describes SVC dumps, including:

- Deciding when to request an SVC dump
- Understanding the types of SVC dumps that MVS produces, and designing your program to handle a specific type
- Identifying the data set to contain the dump
- Defining the contents of the dump
- Suppressing duplicate SVC dumps using dump analysis and elimination (DAE)
- Requesting dumps on other systems in a sysplex
- Using dynamic exits to control dumps on other systems in a sysplex.

For a description of ABEND dumps, see *z/OS MVS Programming: Assembler Services Guide*.

The SDUMPX macro provides the same service as SDUMP, but has the following unique capabilities:

- SDUMPX allows callers in access register (AR) mode to request an SVC dump, whereas SDUMP provides that service only for callers in primary mode.
- SDUMPX allows callers to include data spaces in the SVC dump.
- SDUMPX allows callers to include cross-system coupling facility (XCF) information in the SVC dump.

- SDUMPX allows callers to specify when MVS will post the caller's event control block (ECB) or schedule its service request block (SRB). The calling program can request notification or scheduling after one of the following phases of dump processing:
  - The capture phase; that is, after MVS collects and stores the dump data
  - The write phase; that is, after MVS moves the dump data from storage to an allocated data set.

For simplicity, this information mentions only SDUMPX whenever the SDUMP and SDUMPX services are identical. Functions available only through SDUMPX are noted as such.

For details about coding an SDUMPX or SDUMP macro, see *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

For a description of SVC dumps, see *z/OS MVS Diagnosis: Tools and Service Aids*.

Deciding when to request an SVC dump

Generally, the programs that request dumps of virtual storage are recovery routines, which run in a recovery environment. Under certain circumstances, a recovery routine might not need to request a dump of any storage. For example:

- Another recovery routine might have already requested an SVC dump for this error. If more than one recovery routine might receive control, each routine should determine whether another routine already requested a dump by checking the SDWAEAS bit of the system diagnostic work area (SDWA).
- Some errors do not require a dump for diagnosis. For example, a system completion code of X'913' results when an operator or user does not supply a correct password.

Note: If your program calls a system service (by issuing a macro or callable service), that system service might encounter a user-induced error and end abnormally. Generally, the system does not take dumps for user-induced errors. If you require such a dump, then it is your responsibility to request one in your recovery routine. See Chapter 18, “Providing recovery,” on page 327 for information about writing recovery routines.

When a dump is required for diagnosis, an authorized program can request either an SVC dump or an ABEND dump. Requesting an SVC dump rather than an ABEND dump has several advantages:

- SVC dumps are delivered to the installation's system programmer rather than to the person who ran the abnormally terminating job. Usually, the system programmer wants to see the dumps requested by authorized programs immediately.
- The system processes an SVC dump whether or not the recovery routine percolates or requests a retry. In contrast, the system processes an ABEND dump only if all functional recovery routines (FRRs) percolate, or if no ESTAE-type recovery routine suppresses the dump.
- The system processes an SVC dump closer to the time of error, so the SVC dump might contain more useful data for diagnosis. The system handles an SVC dump request right after the program issues the SDUMPX macro. In contrast, the system handles an ABEND dump request either just before a retry, or after all of the recovery routines have percolated. During percolation, the recovery routines might have altered storage contents or dump options.

Understanding the type of SVC dump that MVS produces

When you design a program to dump storage for diagnostic purposes, you make decisions based on several factors, two of which are: the environment in which the program runs, and the address spaces for which you want storage dumped. These factors determine which parameters you code on the SDUMPX macro, which, in turn, determine the type of SVC dump MVS produces.

- If your program is in supervisor state with PSW key 0, and meets at least one of the following conditions, you can request branch entry into dump processing:
  - The program runs in SRB mode
  - The program holds any lock
  - The program has established an enabled, unlocked task (EUT) FRR.

  To request a branch entry, code BRANCH=YES. A branch entry causes MVS to produce a **scheduled dump**, which offers a potential performance advantage because your program can continue processing as soon as dump processing begins. However, use caution when you code BRANCH=YES; if you do not synchronize your program's processing with dump processing, the dump might not contain valuable data.
- If your program does not meet the environmental restrictions in the preceding item, you must code BRANCH=NO. For BRANCH=NO, the parameters you code to identify storage determine the type of dump MVS produces:
  - If the parameters you coded identify only the primary address space or global data spaces, MVS produces a **synchronous dump**.
  - If the parameters you coded identify address spaces other than, or in addition to, the primary address space, MVS produces a **scheduled dump**.

When MVS produces a synchronous dump, your program cannot run until the capture phase of dump processing completes. However, when MVS produces a scheduled dump, your program can continue its processing as soon as dump processing begins. Because of this difference in processing, you might make different design decisions based on the type of dump MVS produces. To design your program, you need to know which SDUMPX parameter combinations produce each type.

Coding parameters that produce a synchronous dump

MVS produces a synchronous dump when you code BRANCH=NO to request an SVC dump of only the primary address space or global data spaces. **Be careful** if you code the following parameters:

- LISTD=*list addr*. MVS produces a scheduled— not synchronous— dump when the STOKEN represents either an address space that is not the primary address space, or a SCOPE=SINGLE data space owned by a program that is not running in the primary address space.
- SUBPLST=*subpool id list addr*. MVS produces a scheduled— not synchronous— dump when the list of address spaces with associated subpool IDs contains at least one address space other than the primary address space.

Do not code the following parameters if you want a synchronous dump; these parameters automatically cause MVS to produce a scheduled dump:

- ASID=*asid addr*
- ASIDLST
- TYPE=XMEM or TYPE=XMEME
- LISTA.

Designing a program to handle a synchronous dump

With a synchronous dump, MVS usually returns control to your program after the capture phase of dump processing completes. If, however, you code DCB=*dcb addr* to request that MVS use a specific data set for the dump data, MVS returns control after the write phase completes. Your program must then close that data set.

Coding parameters that produce a scheduled dump

MVS produces a scheduled dump if you request branch-entry dump processing, or if you code parameters that identify storage of an address space other than the primary address space, or of multiple address spaces. Specifically, MVS produces a scheduled dump if you code one of the following:

- BRANCH=YES

  To code BRANCH=YES, your program must be in supervisor state with PSW key 0, and also must meet at least one of the following conditions:
  - Is in SRB mode
  - Holds any lock
  - Has an enabled, unlocked task (EUT) FRR established.

  The branch-entry interface uses standard linkage conventions. On entry, general purpose register 13 must point to a 72-byte save area. Branch entry callers must include the CVT mapping macro with the PREFIX=YES parameter.
- BRANCH=NO, with one or more of the following:
  - ASIDLST
  - ASID=*asid addr*
  - TYPE=XMEM or TYPE=XMEME
  - LISTA
  - LISTD=*list addr*, when the STOKEN represents either an address space other than the primary address space, or a SCOPE=SINGLE data space owned by a program that is not running in the primary address space
  - SUBPLST=*subpool id list addr*, when the list of address spaces with associated subpool IDs contains at least one address space other than the primary address space.

Designing a program to handle a scheduled dump

With a scheduled dump, MVS returns control to your program as soon as dump processing begins. In this sense, a scheduled dump is considered an asynchronous dump because your program can continue its processing simultaneously with dump processing.

Because your program can run asynchronously with dump processing, scheduled dumps offer a potential performance advantage. However, under some circumstances, your program might have to synchronize its processing with dump processing by waiting for MVS to capture all of the dump data, or to write the data out to a dump data set.

Synchronizing your program through an ECB or SRB

To synchronize your program with dump processing, you can design the program to use an event control block (ECB) or a service request block (SRB). If you use an ECB, MVS posts the ECB to notify your program that dump processing is complete. If you use an SRB to do some post-dump processing for your program, MVS schedules the SRB after the capture phase or the write phase is complete.

When you synchronize your program with dump processing, MVS usually posts your program's ECB, or schedules its SRB, after the capture phase completes. IBM recommends using the ECB or SRB parameter with CAPTURE instead of WRITE, because your program can continue processing sooner, and resources can be freed sooner. With WRITE, your program not only has to wait for MVS to write the data to a data set, but also might have to wait until an installation operator makes a data set available.

However, in some cases, your program must wait until the write phase completes. MVS posts your program's ECB, or schedules its SRB, after the write phase completes if you:

- Code DCB=*dcb addr* to request that MVS use a specific data set for the dump data. Your program must close that data set after the dump writing phase completes. Whenever you code the DCB parameter, MVS automatically posts your program's ECB or schedules its SRB after the write phase completes, but IBM recommends that you code ECB=(*ecb addr*,WRITE) or SRB=(*srb addr*,WRITE) on SDUMPX to make your source code easier for programmers to understand.
- Use the SDUMPX macro with ECB=(*ecb addr*,WRITE) to request that MVS post your program's ECB when the dump writing phase completes. (If you use the SDUMP macro, you cannot code WRITE on the ECB parameter.)
- Use the SDUMPX macro with SRB=(*srb addr*,WRITE) to request that MVS schedule your program's SRB when the dump writing phase completes. (If you use the SDUMP macro, you cannot code WRITE on the SRB parameter.)

Designing your program to run asynchronously with dump processing

Depending on the tasks your program is designed to accomplish, you might not be able to synchronize it with dump processing. If the environment in which your program runs does not allow you to use an ECB or SRB to synchronize your program, you might lose data that is subject to rapid and frequent change. For example, the data used for communication with other address spaces can change before the system processes a scheduled dump. To avoid this loss, your program can place the data in footprint areas before issuing the SDUMPX macro. Footprint areas include:

- The program's work areas
- The variable recording area (VRA) of the system diagnostic work area (SDWA)
- Any areas you specify through the SUMLIST and SUMLISTA parameters
- Any areas you specify through the SUMLSTL parameter (SDUMPX only)
- The 4K buffer of SQA that is reserved for the caller of SDUMPX, **only** if no other program is using this buffer.

If your program uses footprint areas to save such data, you should provide your installation's programmers with information that explains how your program uses those areas. Otherwise, other programmers might not be able to read the dump or the logrec data set records.

Identifying the data set to contain the dump

By default, MVS places the dump you request on a SYS1.DUMPnn data set. However, you can code the DCB parameter on SDUMPX to specify the data set that you want MVS to use. Because SVC dumps include data from key 0 storage, the data set you use should be made secure. If you use the DCB parameter to specify a data set, your program:

1. Opens the data set
2. Issues SDUMPX with the DCB parameter and, possibly, the ECB or SRB parameter
3. Closes the data set.

If your program uses the DCB and ECB parameters, MVS notifies your program when the write phase completes by posting the ECB. Then your program can close the dump data set. If your program uses the DCB and SRB parameters, MVS schedules the SRB when the write phase completes. As part of its processing, the SRB routine can initiate the process to close the dump data set that your program opened.

If you use SDUMPX to request a scheduled SVC dump, IBM recommends that you code DCB=*dcb addr* with either ECB=(*ecb addr*,WRITE) or SRB=(*srb addr*,WRITE) to have MVS post the ECB or schedule the SRB when the write phase of the dump completes. Whenever you code the DCB parameter, MVS automatically posts your program's ECB or schedules its SRB after the write phase completes, but coding the WRITE option makes your source code easier for programmers to understand.

If you use the SDUMP macro to request an SVC dump, code DCB=*dcb addr* with either ECB=*ecb addr* or SRB=*srb addr*. SDUMP does not recognize the WRITE option. However, MVS automatically posts your program's ECB or schedules its SRB after the write phase completes.

Defining the contents of the dump

Depending on the error that occurs, you might want to tailor the SVC dump to include information that will be the most helpful for diagnosing the error. Through the SDUMPX parameters, you can:

- Identify the address space or spaces to be dumped
- Identify the data spaces to be dumped (available through SDUMPX only)
- Customize the contents of the dump for those address spaces
- Request a summary dump.

Identifying the address spaces or data spaces to be dumped

You can request an SVC dump of the virtual storage for a maximum of 15 address spaces.

- To dump the storage of one address space, code ASID=*asid addr*. If you omit the ASID parameter, the SVC dump will contain the storage of the primary address space (the default). If you code ASID=0, the SVC dump will contain the storage of the home address space.
- To dump the storage of multiple address spaces, code ASIDLST=*list addr*.
- To allow the cross-memory mode of the calling program to determine which address spaces to dump, code TYPE=XMEME or TYPE=XMEM. With these parameters, you request an SVC dump of the home, primary, and secondary address spaces, from either the time of the error or the time of the dump request.
- To dump ranges of storage within address spaces, code LISTA=*list addr*.
- To dump subpool storage associated with address spaces, code SUBPLST=*subpool id list addr*.
- To dump ranges of referenced storage in address or data spaces, code LISTD=*list addr* on SDUMPX only.

Customizing the contents of the SVC dump

For a specific address space, or for each address space you identify, you can request that the system include or exclude specific storage or ranges of storage. By coding additional parameters, you can request that the system:

- Dump certain areas of private or common storage (SDATA parameter with options)
- Override default options (SDATA parameter with NODEFS).

Do not specify all SDATA parameters unless you need all storage areas to diagnose the error. If you do not need a particular area, omit the corresponding option or omit the SDATA parameter. Whenever possible, tailor the SVC dump using the storage list parameters, such as LISTD (on SDUMPX only), LIST, LISTA, and STORAGE.

Generally, each installation defines default options for SVC dumps and sets up specific formatters for those dumps. If you override the default options, provide your own interactive problem control system (IPCS) CLIST to format the dump contents.

Requesting the summary dump

The dump you request through SDUMPX contains a summary dump by default. A summary dump, part of an SVC dump, is formatted and contains the storage of certain system data areas. Although the summary dump is a default, you can explicitly request the system to include one by coding SDATA=SUMDUMP.

The type of summary dump that MVS includes depends on the SDUMPX parameters you code:

Table 63. SDUMPX Parameters and the Type of Summary Dump They Produce

If you code:	**The summary dump type is:**
BRANCH=YES, SUSPEND=NO	Disabled

BRANCH=YES, SUSPEND=YES
or
BRANCH=NO, SUMLSTL=*list addr*
	Suspend

BRANCH=NO (SUSPEND cannot be specified)	Enabled

Descriptions of each type of summary dump follow.

Disabled Summary Dump
In a disabled summary dump, MVS captures system data that is subject to rapid and frequent change. Because the system is disabled, MVS can capture data from only paged-in or disabled reference (DREF) storage. The caller can specify the SUMLIST and SUMLSTA parameters on SDUMP or SDUMPX and the SUMLSTL parameter on SDUMPX to save specific data in the summary dump.

Suspend Summary Dump
In a suspend summary dump, MVS also captures system data that is subject to rapid and frequent change. However, MVS can capture data from paged-in, DREF, and pageable storage. The caller can specify the SUMLIST and SUMLSTA parameters on SDUMP or SDUMPX and the SUMLSTL parameter on SDUMPX to save specific data in the summary dump.

Enabled Summary Dump

The purpose of the enabled summary dump is to group data for debugging dumps by specifying a particular option on the SDUMP or SDUMPX macro. If the dump is a scheduled dump, the summary data is saved for each address space specified.

Suppressing SVC dumps that duplicate previous SVC dumps

Where To Find Prerequisite Information: The dump and analysis elimination (DAE) function is available to both unauthorized and authorized programs. Before reading the following topic, read about DAE and about how to provide symptoms through the SDWA in *z/OS MVS Programming: Assembler Services Guide*. See also *z/OS MVS Diagnosis: Tools and Service Aids* for starting and controlling dump suppression by DAE.

Providing symptom information through the SDUMPX macro

The SYMREC parameter on the SDUMPX macro allows programs running in a non-recovery environment, where there is no SDWA, to request an SVC dump and dump suppression services similar to those that are available in a recovery environment, where an SDWA is present. When an SDWA exists **and** a symptom record is passed to the system on the SYMREC parameter, DAE uses the primary symptom string in the symptom record. DAE suppresses the SVC dump if the primary symptom string in the symptom record matches previously known symptoms.

DAE copies the primary and secondary symptom strings from the symptom record into the dump header. If the symptom record is written to the logrec data set, the symptom string in the dump header is consistent with the record in the logrec data set. The system does not include the symptom record in the dump, but you can include the symptom record in the dump by using the SUMLIST keyword on the SDUMPX macro.

Consider the following points when using the SYMREC parameter of the SDUMPX macro to specify a symptom record:

- The caller must build the symptom record using the SYMRBLD macro, or the ADSR mapping macro, and fill in at least the ‘SR’ identifier and the primary symptom string, which should uniquely identify the error.

  If the symptom record identifier is not ‘SR’ or if the symptoms are not addressable, the system issues an abend with a completion code of X'233', then returns to the caller with a return code of 8.

  See *z/OS MVS Programming: Assembler Services Guide* for instructions and programming notes for using the SYMRBLD macro or the ADSR mapping macro.
- DAE uses only the first 150 bytes of the primary symptom string in the symptom record. If it must truncate a primary symptom string, DAE truncates at the end of the last complete symptom within the first 150 bytes of the symptom string.
- During its match processing, DAE performs a byte-by-byte comparison of symptom strings. Thus, be sure that the symptom strings you generate are consistent in the order of symptoms and the number of blank characters between symptoms.
- If the SYMREC parameter is used in a recovery environment where an SDWA exists, DAE uses the symptoms found in the symptom record rather than in the SDWA.

Requesting dumps on other systems in a sysplex

In a sysplex, a problem can involve several or all of the systems. The SDUMPX macro can request an SVC dump on the local system and, using the REMOTE parameter, an SVC dump on one or more of the other systems in the sysplex. Using the parameters on the macro, you can make the dump requests the same or different, depending on the areas you think a diagnostician will need for any problems.

In systems in a sysplex, the installation should specify names that form patterns so that all the names can be specified in one value that contains wildcards. In the SDUMPX macro, you can use wildcards, * and ?, to identify all or some names, based on the patterns for the names. For example, jobs can be named TRANS1, TRANS2, TRANS3, ... , TRANS17. A jobname of TRANS* in the SDUMPX macro will dump all of them. In the SDUMPX macro, you can use wildcards in system names, address space identifiers, job names, XCF groups, XCF members, and data space names.

Using dynamic exits to control dumps in a sysplex

Three types of dynamic exits control SVC dumps requested from another system in a sysplex. The exits are:

- IEASDUMP.QUERY - SVC dump query exit
- IEASDUMP.GLOBAL - SVC dump exit for capturing common storage
- IEASDUMP.LOCAL - SVC dump exit for capturing storage in its address space
- IEASDUMP.SERVER - SVC dump exit for adding data to the dump.

IEASDUMP.QUERY dynamic exit

You can use an IEASDUMP.QUERY exit:

- To decide if an SVC dump requested by another system in the sysplex is needed. To indicate that this system should be dumped, do the following:
  - Specify a SDUMPX macro with a REMOTE parameter
  - Place SYSDCOND in the PROBDESC area for the macro
  - Pass a return code of 0 from at least one IEASDUMP.QUERY exit

  To indicate that this system should **not** be dumped, SYSDCOND must be in the PROBDESC area and either of the following must be true:
  - No IEASDUMP.QUERY exit exists
  - No IEASDUMP.QUERY exit returns a code of 0
- To identify storage areas to be added to dumps that are needed.

Installing IEASDUMP.QUERY

Use the CSVDYNEX macro to connect the routine to the SVC dump processing. For example:

```
CSVDYNEX REQUEST=ADD,
        EXITNAME=ROUTQ,
        MODNAME=MODQ,
        STOKEN=(3)
.
.
.
ROUTQ      DC CL16'IEASDUMP.QUERY'
MODQ       DC CL8'SDUMPQ'
```

IEASDUMP.QUERY dynamic exit environment

The exit receives control in the following environment:

- In supervisor state with PSW key 0.
- In dispatchable unit mode of SRB.
- In cross memory mode of PASN=HASN=SASN.
- In AMODE 31.
- In primary ASC mode.
- Enabled for I/O and external interrupts.
- With no locks held.
- With parameter areas in the primary address space. The parameter area for IEASDUMP.QUERY is mapped by the IHASDMSE mapping macro.

IEASDUMP.QUERY dynamic exit recovery

The exit should provide its own recovery, using a functional recovery routine (FRR). If the IEASDUMP.QUERY exit ends abnormally, its recovery routine will get control first.

You can use the ADDABENDNUM and ABENDCONSEC parameters on the CSVDYNEX REQUEST=ADD macro to limit the number of times the exit routine abnormally ends before it becomes inactive. An abend is counted under the following conditions:

- The exit routine does not provide recovery, or the exit routine does provide recovery but percolates the error.
- The system allows a retry, that is, the recovery routine is entered with bit SDWACLUP off.

By default, the system does not disable the exit routine.

If an IEASDUMP.QUERY exit ends abnormally, the system will proceed as though the exit had returned a nonzero return code. If it is the only IEASDUMP.QUERY exit invoked, no dump will be written. If other IEASDUMP.QUERY exits are invoked, a dump will be written if one of the other exits returns a code of 0. If all other IEASDUMP.QUERY exits return nonzero codes or abnormally end and SYSDCOND is in the PROBDESC area, no dump will be written.

IEASDUMP.QUERY dynamic exit processing

All IEASDUMP.QUERY exits that are defined receive control when another system in the sysplex requests an SVC dump on this system using:

- An SDUMPX macro with a REMOTE parameter and with or without a PROBDESC parameter.
- A DUMP command with a REMOTE parameter and with a PROBDESC parameter.

The IEASDUMP.QUERY exit on the *local system*, which is the system requesting the dumps, also receives control if PROBDESC=SYSDLOCL is specified and if the local system is included in the systems specified by SYSLIST or GRPLIST through the REMOTE parameter.

The IEASDUMP.QUERY exit receives control before the system obtains the areas to be dumped. The exit receives control in one of the following address spaces:

- **In the address space represented by the STOKEN parameter**, if STOKEN was specified on the CSVDYNEX REQUEST=ADD macro that installed the routine.

- **In the address space represented by the job specified in the JOBNAME parameter**, if JOBNAME was specified on the CSVDYNEX REQUEST=ADD macro. If several jobs have the specified job name, the exit gets control in only one of the address spaces, unless another of those address spaces is also identified by the STOKEN on another of the CSVDYNEX REQUEST=ADD macros. For example:
  - CSVDYNEX REQUEST=ADD for exit R1 and with STOKEN=x, which is for ASID 002F.
  - CSVDYNEX REQUEST=ADD for exit R2 and with JOBNAME=J. Both ASID 002E and 002F are for the job named J.

  Exit R1 gets control only in ASID 002F. Exit R2 gets control in both ASID 002E and 002F. However, if R1 had not been associated with this exit, R2 would get control only in ASID 002E. To best limit the address spaces in which the exit will run, use STOKEN rather than JOBNAME.
- **In address space ASID 0001 and in the address spaces of any other IEASDUMP.QUERY exits, based on their STOKEN and JOBNAME values**, if neither STOKEN nor JOBNAME was specified.

IBM recommends that you specify an STOKEN parameter for an IEASDUMP.QUERY exit.. To obtain the STOKEN for the home address space, issue an ALESERV macro with an EXTRACTH parameter. To obtain the STOKEN for ASID 0001:

- Issue the macro `LOCASCB ASID==H'1'` to locate the address space control block (ASCB) for ASID 0001. (The two equal signs are needed.)
- Use the ASCBASSB field in the ASCB to locate the address space secondary block (ASSB) for ASID 0001.
- Obtain the STOKEN for ASID 0001 from the ASSBSTKN field of the ASSB.

The IEASDUMP.QUERY exit decides if its system should write an SVC dump. The IEASDUMP.QUERY exit returns control with a return code to indicate the actions to be taken.

If the return code indicates that a dump should be written, the exit can add areas to be dumped, specifying them in the area mapped by the IHASDMSE mapping macro. For example:

- Address spaces, specified in an area mapped by DSECT SDMSE_ASIDLST. When used, place the address of this area in SDMSE_OUTPUT_ASIDLST_ADDR.
- Storage ranges, specified in an area mapped by DSECT SDMSE_STORAGE. When used, place the address of this area in SDMSE_OUTPUT_STORAGE_ADDR.
- Jobs, specified in an area mapped by DSECT SDMSE_JOBLIST. When used, place the address of this area in SDMSE_OUTPUT_JOBLIST_ADDR.
- Data spaces, specified in an area mapped by DSECT SDMSE_DSPLIST. When used, place the address of this area in SDMSE_OUTPUT_DSPLIST_ADDR.
- Subpools, specified in an area mapped by DSECT SDMSE_SUBPLST. When used, place the address of this area in SDMSE_OUTPUT_SUBPLST_ADDR.
- Keys for the subpools, specified in an area mapped by DSECT SDMSE_KEYLIST. When used, place the address of this area in SDMSE_OUTPUT_KEYLIST_ADDR.
- SDATA options, specified in the SDMSE_SDATA_OPTIONS fields.

The exit can place data to be dumped in a 4096-byte workarea supplied by the system; the address of the workarea is in the SDMSE_INPUT_WORKARE_ADDR field. If the exit requires more storage, the exit should request it with a GETMAIN or STORAGE OBTAIN macro and describe it as follows:

- Its address in the SDMSE_OUTPUT_WORKAREA_ADDR field.
- Its length in the SDMSE_OUTPUT_WORKAREA_LENGTH field.
- Its subpool in the SDMSE_OUTPUT_WORKAREA_SUBPOOL field.
- Its key in the SDMSE_OUTPUT_WORKAREA_KEY field. Make sure that the key is in bits 0-3 of the 8-bit field. For example, if using key 1, set the field to X'10'.

Upon return from the routine, the system releases the storage described by these fields.

Code the IEASDUMP.QUERY exit to be reentrant.

Immediate and Deferred Local Dump: For the local system that requests the dumps, you can obtain two related dumps: an immediate dump containing current problem data and a deferred dump containing areas added by the IEASDUMP exits.

Specify the immediate and deferred dumps as follows:

- Specify the local system in SYSLIST or GRPLIST through the REMOTE parameter of the DUMP command or SDUMPX macro. You can specify the local system explicitly or with wildcards. For example, an * indicates all systems. SY? indicates the local system if it is named, for example, SY1.
- Specify SYSDCOND and SYSDLOCL keys on the PROBDESC parameter of the DUMP command or in the PROBDESC area for the SDUMPX macro.

When you specify the preceding for four systems, SY1, SY2, SY3, and SY4, with SY1 as the system requesting the dumps, the results are:

- An SVC dump is requested on SY1.
- SY1 requests dumps on SY2, SY3, and SY4, and also on SY1.
- On SY2, SY3, and SY4, dump processing begins. If any IEASDUMP.QUERY exit returns 0, a dump is written for the system.
- On SY1, dump processing also begins. If any IEASDUMP.QUERY exit returns 0, a dump is written.

The total dumps requested are five, two on system SY1 and one each on systems SY2, SY3, and SY4.

Registers at entry to IEASDUMP.QUERY dynamic exit

The contents of the registers on entry to an exit are:

Register
Contents

0 Not applicable

1 Address of the parameter area mapped by the IHASDMSE mapping macro.

2-12 Not applicable

13 Address of the 72-byte standard save area

14 Return address

15 Entry point address of IEASDUMP.QUERY.

Parameter area at entry to IEASDUMP.QUERY dynamic exit

The parameter area is mapped by the IHASDMSE mapping macro. For IHASDMSE, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). Information that the area provides to the IEASDUMP.QUERY exit includes the problem description provided through the PROBDESC parameter.

Registers at exit from IEASDUMP.QUERY dynamic exit

Upon return from IEASDUMP.QUERY processing, the register contents must be:

Register
Contents

0-1 Not applicable

2-13 Restored to contents at entry

14 Not applicable

15 One of the following return codes:

Return Code
Explanation

0 Write the requested dump. Dump added areas, if specified in the parameter list, are mapped by the IHASDMSE mapping macro.

4 Do not add areas to the dump. If the DUMP command or SDUMPX macro contains PROBDESC=SYSDCOND, suppress the dump of this system.

Disassociating IEASDUMP.QUERY

Disassociate the exit from SVC dump when it should no longer receive control because the task or address space ended. Use the CSVDYNEX macro to disassociate the exits. For example:

```
CSVDYNEX REQUEST=DELETE,
         EXITNAME=ROUTQ,
         MODNAME=MODQ
.
.
.
ROUTQ      DC CL16'IEASDUMP.QUERY'
MODQ       DC CL8'SDUMPQ'
```

Disassociation is needed to prevent the routine from being invoked after the application is ended, and to release any common storage that might be used for it.

IEASDUMP.GLOBAL and IEASDUMP.LOCAL dynamic exits

You can use the IEASDUMP.GLOBAL and IEASDUMP.LOCAL exits dentify storage areas to be added to an SVC dump requested by another system in the sysplex.

Installing IEASDUMP.GLOBAL and IEASDUMP.LOCAL

Use the CSVDYNEX macro to connect each exit to the SVC dump processing. For example:

```
CSVDYNEX REQUEST=ADD,
         EXITNAME=ROUTGLO,
         MODNAME=MODGLO
CSVDYNEX REQUEST=ADD,
         EXITNAME=ROUTLOC,
         MODNAME=MODLOC
         JOBNAME=ANY
.
.
.
ROUTGLO    DC CL16'IEASDUMP.GLOBAL'
MODGLO     DC CL8'SDUMPGLO'
ROUTLOC    DC CL16'IEASDUMP.LOCAL'
MODLOC     DC CL8'SDUMPLOC'
```

Use JOBNAME or STOKEN to limit further the address spaces in which IEASDUMP.LOCAL exits receive control.

IEASDUMP.GLOBAL and IEASDUMP.LOCAL dynamic exit environment

The exits receive control in the following environment:

- In supervisor state with PSW key 0.
- In dispatchable unit mode of task.
- In cross memory mode of PASN=HASN=SASN.
- In AMODE 31.
- In primary ASC mode.
- Enabled for I/O and external interrupts.
- With no locks held.
- With parameter areas in the primary address space. The parameter area for IEASDUMP.GLOBAL or IEASDUMP.LOCAL is mapped by the IHASDEXI mapping macro.

IEASDUMP.GLOBAL and IEASDUMP.LOCAL dynamic exit

Each exit should provide its own recovery, using an ESTAEX or ESTAE macro. If the IEASDUMP.GLOBAL or IEASDUMP.LOCAL exit ends abnormally, its recovery routine will get control first.

You can use the ADDABENDNUM and ABENDCONSEC parameters on the CSVDYNEX REQUEST=ADD macro to limit the number of times the exit routine abnormally ends before it becomes inactive. An abend is counted under the following conditions:

- The exit routine does not provide recovery, or the exit routine does provide recovery but percolates the error.
- The system allows a retry, that is, the recovery routine is entered with bit SDWACLUP off.

By default, the system does not disable the exit routine.

If a IEASDUMP.GLOBAL or IEASDUMP.LOCAL exit ends abnormally or times out, a partial dump is written. The dump will indicate that an error occurred in an IEASDUMP exit routine.

IEASDUMP.GLOBAL and IEASDUMP.LOCAL dynamic exit processing

The IEASDUMP.GLOBAL and IEASDUMP.LOCAL exits, if defined, receive control when another system in the sysplex requests an SVC dump on this system and the IEASDUMP.QUERY did not block the dump. The SVC dump on this system is requested by:

- A SDUMPX macro with a REMOTE parameter
- A DUMP command with a REMOTE parameter

The exits receive control as follows:

- IEASDUMP.GLOBAL exit: Receives control in the DUMPSRV address space during global data capture, once for an SVC dump on its system.

  Note: Do not specify STOKEN on the CSVDYNEX REQUEST=ADD macro for IEASDUMP.GLOBAL.
- IEASDUMP.LOCAL exit: Receives control during local data capture in each address space being dumped for an SVC dump on its system. The exit can run several times for one dump.

  If STOKEN is specified in the CSVDYNEX REQUEST=ADD macro for an IEASDUMP.LOCAL, the exit will receive control only in the address space represented by the STOKEN and only when the dump includes that address space.

Both exits can add areas to be dumped, specifying them in the area mapped by the IHASDEXI mapping macro. You can dump a single page of data or, if a range is specified, pages of data:

- Indicate a range by turning on bit SDEXIADDRESSRANGE and providing the end address in field SDEXIRANGEEND. See the description of bit SDEXIADDRESSRANGE for more complete information about providing an address range.
- Indicate a single page of data by making sure that bit SDEXIADDRESSRANGE is off. Place the 4096 bytes of data in the buffer whose address is in field SDEXIFAD.

Note that, in all cases, the system dumps entire 4096-byte pages. The addresses you specify are rounded down or up to page boundaries, as needed.

If the storage to be dumped is in the primary address space, specify its start address in field SDEXICDAD. If the storage is not in the primary address space, define a dump record prefix, which describes the location and the address space or data space being dumped. When defining a dump record prefix, set bit SDEXIDRPS on and place in field SDEXIDRPXLAD the address of the start of the area to be included in the dump; for example, the start of a range, or the address of the 4096-byte buffer's contents, or the address to be associated with the component data. The dump record prefix is a 64-byte area provided by the system, mapped by DSECT SDEXIDRPX, and pointed to by field SDEXIDRPA. See the description preceding the SDEXIDRPX DSECT for information about the fields to set in the dump record prefix area.

After setting the fields in the area mapped by IHASDEXI, the IEASDUMP.GLOBAL or IEASDUMP.LOCAL exit should call the system routine whose address is in the SDEXIORAD field. The system routine adds the storage range or buffer to the dump.

The IEASDUMP.GLOBAL or IEASDUMP.LOCAL exit is to use the following interface when calling the system routine. The system routine must receive control in the following environment:

- In supervisor state with PSW key 0.
- In dispatchable unit mode of task.
- In cross memory mode of any PASN, any SASN, and the HASN under which the IEASDUMP exit received control.
- In AMODE 31.

- In primary ASC mode.
- Enabled for I/O and external interrupts.
- With no locks held.
- With no enabled, unlocked task (EUT) FRRs.

The contents of the registers on entry to the system routine addressed by SDEXIORAD are:

Register	**Contents**
0	Not applicable
1	Address of SDEXI
2-12	Not applicable
13	Address of a 72-byte standard save area
14	Return address
15	Entry point address

The contents of the registers on exit from the system routine addressed by SDEXIORAD are:

Register	**Contents**
0-1	Used as work registers
2-13	Unchanged
14-15	Used as work registers

Code the IEASDUMP.GLOBAL and IEASDUMP.LOCAL exits to be reentrant.

Registers at entry to IEASDUMP.GLOBAL or IEASDUMP.LOCAL dynamic exit

The contents of the registers on entry to a IEASDUMP.GLOBAL or IEASDUMP.LOCAL exit are:

Register	**Contents**
0	Not applicable
1	Address of the parameter area mapped by the IHASDEXI mapping macro.
2-12	Not applicable
13	Address of the 72-byte standard save area
14	Return address
15	Entry point address of IEASDUMP.GLOBAL or IEASDUMP.LOCAL

Parameter area at entry to IEASDUMP.GLOBAL or IEASDUMP.LOCAL dynamic exit

The parameter area is mapped by the IHASDEXI mapping macro. For IHASDEXI, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). Information that the area provides to the IEASDUMP.GLOBAL or IEASDUMP.LOCAL exit includes the problem description provided through the PROBDESC parameter and the incident token for the related dumps. The area gives the address and length of the data buffer the exit should use.

Registers at exit from IEASDUMP.GLOBAL or IEASDUMP.LOCAL dynamic exit

Upon return from IEASDUMP.GLOBAL or IEASDUMP.LOCAL processing, the register contents must be:

Register
Contents

0-1 Not applicable

2-13 Restored to contents at entry

14 Not applicable

15 One of the following return codes:

Return Code
Explanation

0 Exit processed successfully. Data was added to the dump.

4 Exit did not process successfully. Some data was not added to the dump. Upon dump completion, dump processing will issue a partial-dump message.

Disassociating IEASDUMP.GLOBAL and IEASDUMP.LOCAL

Disassociate the exits from SVC dump when they should no longer receive control because the task or address space ended. Use the CSVDYNEX macro to disassociate the exits. For example:

```
CSVDYNEX REQUEST=DELETE,
         EXITNAME=ROUTGLO,
         MODNAME=MODGLO
CSVDYNEX REQUEST=DELETE,
         EXITNAME=ROUTLOC,
         MODNAME=MODLOC
.
.
.
ROUTGLO    DC CL16'IEASDUMP.GLOBAL'
MODGLO     DC CL8'SDUMPGLO'
ROUTLOC    DC CL16'IEASDUMP.LOCAL'
MODLOC     DC CL8'SDUMPLOC'
```

Disassociation is needed to prevent the routine from being invoked after the application is ended, and to release any common storage that might be used for it.

IEASDUMP.SERVER dynamic exit

You can use an IEASDUMP.SERVER exit to identify data to be added to the dump. The current dump request is provided in register 1 and is mapped by IHASDMSE. Exit routines should scan the current dump request to determine if data should be added to the dump.

Installing IEASDUMP.SERVER

Use the CSVDYNEX macro to connect the exit to the SVC dump processing. For example:

```
CSVDYNEX REQUEST=ADD,
         EXITNAME=ROUTQ,
         MODNAME=MODQ,
        STOKEN=(3)
.
.
.
ROUTQ      DC CL16'IEASDUMP.SERVER'
MODQ       DC CL8'SDUMPQ'
```

IEASDUMP.SERVER dynamic exit environment

The exit receives control in the following environment:

- In supervisor state with PSW key 0
- In dispatchable unit mode of TCB
- In cross memory mode of PASN=HASN=SASN
- In AMODE 31
- In primary ASC mode
- Enabled for I/O and external interrupts
- With no locks held
- With parameter areas in the primary address space. The parameter area for IEASDUMP.SERVER is mapped by the IHASDMSE mapping macro.

IEASDUMP.SERVER dynamic exit recovery

The exit should provide its own recovery using an EXTAE(X) routine. If the IEASDUMP.SERVER exit ends abnormally, its recovery routine will get control first.

You can use the ADDABENDNUM and ABENDCONSEC parameters on the CSVDYNEX REQUEST=ADD macro to limit the number of times the exit routine abnormally ends before it becomes inactive. An abend is counted under the following conditions:

- The exit routine does not provide recovery, or the exit routine does provide recovery but percolates the error.
- The system allows a retry; that is, the recovery routine is entered with bit SDWACLUP off.

By default, the system does not disable the exit routine.

If an IEASDUMP.SERVER exit ends abnormally, the system will proceed as though the exit had returned a nonzero return code. If it is the only IEASDUMP.SERVER exit invoked, no data will be added. If other IEASDUMP.SERVER exits are invoked, data will be added if one of the other exits returns a code of 0. If all other IEASDUMP.SERVER exits return nonzero codes or abnormally end, no data will be added.

IEASDUMP.SERVER dynamic exit processing

All IEASDUMP.SERVER exits that are defined receive control when SDATA=SERVERS are specified on the SDUMPX macro, on the DUMP command, or on an SDUMP option of the CHNGDUMP SDUMP command.

The IEASDUMP.SERVER exit receives control before the system obtains the areas to be dumped. The exit receives control in the DUMPSRV address space.

The current dump request is provided to the caller and is mapped by the IHASDMSE mapping macro. For example:

- Address spaces, specified in an area mapped by DSECT SDMSE_ASIDLST. The address of this area is in SDMSE_INPUT_ASIDLST_ADDR.

- Storage ranges, specified in an area mapped by DSECT SDMSE_STORAGE. The address of this area is in SDMSE_INPUT_STORAGE_ADDR.
- Subpools, specified in an area mapped by DSECT SDMSE_SUBPLST. The address of this area is in SDMSE_INPUT_SUBPLST_ADDR.
- Keys for the subpools, specified in an area mapped by DSECT SDMSE_KEYLIST. The ddress of this area is in SDMSE_INPUT_KEYLIST_ADDR.
- SDATA options, specified in the SDMSE_SDATA_OPTIONS fields.

If the return code indicates that data should be added, the exit can add the areas to be dumped, specifying them in the area mapped by the IHASDMSE mapping macro. For example:
- Address spaces, specified in an area mapped by DSECT SDMSE_ASIDLST. When used, place the address of this area in SDMSE_OUTPUT_ASIDLST_ADDR.
- Storage ranges, specified in an area mapped by DSECT SDMSE_STORAGE. When used, place the address of this area in SDMSE_OUTPUT_STORAGE_ADDR.
- Jobs, specified in an area mapped by DSECT SDMSE_JOBLIST. When used, place the address of this area in SDMSE_OUTPUT_JOBLIST_ADDR.
- Data spaces, specified in an area mapped by DSECT SDMSE_DSPLIST. When used, place the address of this area in SDMSE_OUTPUT_DSPLIST_ADDR.
- Subpools, specified in an area mapped by DSECT SDMSE_SUBPLST. When used, place the address of this area in SDMSE_OUTPUT_SUBPLST_ADDR.
- Keys for the subpools, specified in an area mapped by DSECT SDMSE_KEYLIST. When used, place the address of this area in SDMSE_OUTPUT_KEYLIST_ADDR.
- SDATA options, specified in the SDMSE_SDATA_OPTIONS fields.

The exit can place data to be dumped in a 4096-byte work area supplied by the system. The address of the work area is in the SDMSE_INPUT_WORKAREA_ADDR field. If the exit requires more storage, the exit should request it with a GETMAIN or STORAGE OBTAIN macro and describe it as follows:
- Address in the SDMSE_OUTPUT_WORKAREA_ADDR field
- Length in the SDMSE_OUTPUT_WORKAREA_LENGTH field
- Subpool in the SDMSE_OUTPUT_WORKAREA_SUBPOOL field
- Key in the SDMSE_OUTPUT_WORKAREA_KEY field. Make sure that the key is in bits 0-3 of the 8-bit field. For example, if using key 1, set the field to X'10'.

Upon return from the exit, the system releases the storage described by these fields.

Code the IEASDUMP.SERVER exit to be reentrant.

Registers at entry to IEASDUMP.SERVER dynamic exit

The contents of the registers on entry to an exit are:

Register
Contents

0 Not applicable

1 Address of the parameter area mapped by the IHASDMSE mapping macro

2-12 Not applicable

13 Address of the 72-byte standard save area

14 Return address

15 Entry point address of IEASDUMP.SERVER

Parameter area at entry to IEASDUMP.SERVER dynamic exit

The parameter area is mapped by the IHASDMSE mapping macro. For IHASDMSE, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). Information that the area provides to the IEASDUMP.SERVER exit includes the problem description provided through the PROBDESC parameter.

Registers at exit from IEASDUMP.SERVER dynamic exit

Upon return from IEASDUMP.SERVER processing, the register contents must be:

Register
Contents

0-1 Not applicable

2-13 Restored to contents at entry

14 Not applicable

15 One of the following return codes:

Return Code
Explanation

0 Add the requested data to the dump. Dump added areas, if specified in the parameter list, are mapped by the IHASDMSE mapping macro.

4 Do not add areas to the dump.

Disassociating IEASDUMP.SERVER

Disassociate the exit from SVC dump when it should no longer receive control because the task or address space ended. Use the CSVDYNEX macro to disassociate the exits. For example:

```
CSVDYNEX REQUEST=DELETE,
         EXITNAME=ROUTQ,
         MODNAME=MODQ
.
.
.
ROUTQ      DC CL16'IEASDUMP.SERVER'
MODQ       DC CL8'SDUMPQ'
```

Disassociation is needed to prevent the routine from being invoked after the application is ended, and to release any common storage that might be used for it.

Transaction dumps

This information describes Transaction dumps, including:

- Deciding when to request a Transaction dump
- Understanding the types of Transaction dumps that MVS produces, and designing your program to handle a specific type
- Identifying the data set to contain the dump
- Defining the contents of the dump

- Suppressing duplicate Transaction dumps using dump analysis and elimination (DAE)
- Requesting dumps of other address spaces on the same system or on other systems in the same sysplex

Deciding when to request a transaction dump

Generally, the programs that request dumps of virtual storage are recovery routines, which run in a recovery environment. Under certain circumstances, a recovery routine might not need to request a dump of any storage. For example:

- Another recovery routine might have already requested a Transaction dump for this error. If more than one recovery routine might receive control, each routine should determine whether another routine already requested a dump by checking the SDWAEAS bit of the system diagnostic work area (SDWA).
- Some errors do not require a dump for diagnosis. For example, a system completion code of X'913' results when an operator or user does not supply a correct password.

Note: If your program calls a system service (by issuing a macro or callable service), that system service might encounter a user-induced error and end abnormally. Generally, the system does not take dumps for user-induced errors. If you require such a dump, then it is your responsibility to request one in your recovery routine. See Chapter 18, “Providing recovery,” on page 327 for information about writing recovery routines.

When a dump is required for diagnosis, both unauthorized and authorized programs can request either a Transaction dump or an ABEND dump. Requesting a Transaction dump rather than an ABEND dump has several advantages:

- The system processes a Transaction dump whether or not the recovery routine percolates or requests a retry. In contrast, the system processes an ABEND dump only if all functional recovery routines (FRRs) percolate, or if no ESTAE-type recovery routine suppresses the dump.
- The system processes a Transaction dump closer to the time of error, so the Transaction dump might contain more useful data for diagnosis. The system handles a Transaction dump request right after the program issues the IEATDUMP macro. In contrast, the system handles an ABEND dump request either just before a retry, or after all of the recovery routines have percolated. During percolation, the recovery routines might have altered storage contents or dump options.

Understanding the type of transaction dump that MVS produces

When you design a program to dump storage for diagnostic purposes, you make decisions based on several factors, two of which are: the environment in which the program runs, and the storage you want dumped. These factors determine which parameters you code on the IEATDUMP macro, which, in turn, determine the type of Transaction dump MVS produces.

Identifying the data set to contain the dump

You can request that the Transaction dump be written to a data set that is either pre-allocated or automatically allocated. To request a pre-allocated data set, specify the DDNAME parameter that identifies a data set that contains sufficient space in one or more extents for the entire dump to be written. If you don't provide a large enough data set, you will receive a partial dump. To request automatic allocation, specify the DSN and DSNAD parameters to diminish the exposure that the dump is truncated because of data set space constraints. Automatic allocation is done to SYSALLDA. When using DSN or DSNAD, the maximum dump size is 2 GB. To collect data beyond that, use the &DS symbol at the end of the pattern. Then, the dump data is split across multiple data sets. DDNAME does not have the 2GB size restriction.

Defining the contents of the dump

Depending on the error that occurs, you might want to tailor the Transaction dump to include information that will be the most helpful for diagnosing the error. Through the IEATDUMP parameters, you can:

- Identify the address space or spaces to be dumped
- Identify the data spaces to be dumped
- Customize the contents of the dump for those address spaces
- Request a summary dump.

Customizing the contents of the transaction dump

For a specific address space, or for each address space you identify, you can request that the system include or exclude specific storage or ranges of storage. By coding additional parameters, you can request that the system dump certain areas of private or common storage (SDATA parameter with options).

Do not specify all SDATA parameters unless you need all storage areas to diagnose the error. If you do not need a particular area, omit the corresponding option or omit the SDATA parameter.

Generally, each installation defines default options for Transaction dumps and sets up specific formatters for those dumps. If you override the default options, provide your own interactive problem control system (IPCS) CLIST to format the dump contents.

Requesting the summary dump

The purpose of the summary dump is to group data for debugging dumps. By specifying SDATA= with the REMOTE parameter on the IEATDUMP macro, summary data is saved for each address space specified.

Suppressing transaction dumps that duplicate previous transaction dumps

Where To Find Prerequisite Information: The dump and analysis elimination (DAE) function is available to both unauthorized and authorized programs. Before reading the following topic, read about DAE and about how to provide symptoms through the SDWA in *z/OS MVS Programming: Assembler Services Guide*. See also *z/OS MVS Diagnosis: Tools and Service Aids* for starting and controlling dump suppression by DAE.

Providing symptom information through the IEATDUMP macro

The SYMREC parameter on the IEATDUMP macro allows programs running in a non-recovery environment, where there is no SDWA, to request a Transaction dump and dump suppression services similar to those that are available in a recovery environment, where an SDWA is present. When an SDWA exists **and** a symptom record is passed to the system on the SYMREC parameter, DAE uses the primary symptom string in the symptom record. DAE suppresses the Transaction dump if the primary symptom string in the symptom record matches previously known symptoms.

DAE copies the primary and secondary symptom strings from the symptom record into the dump header. If the symptom record is written to the logrec data set, the symptom string in the dump header is consistent with the record in the logrec data set. The system does not include the symptom record in the dump, but you can include the symptom record in the dump by using the PROBDESC keyword on the IEATDUMP macro.

Consider the following points when using the SYMREC parameter of the IEATDUMP macro to specify a symptom record:

- The caller must build the symptom record using the SYMRBLD macro, or the ADSR mapping macro, and fill in at least the 'SR' identifier and the primary symptom string, which should uniquely identify the error.

  If the symptom record identifier is not 'SR' or if the symptoms are not addressable, the system returns to the caller with a return code of 8.

  See *z/OS MVS Programming: Assembler Services Guide* for instructions and programming notes for using the SYMRBLD macro or the ADSR mapping macro.
- DAE uses only the first 150 bytes of the primary symptom string in the symptom record. If it must truncate a primary symptom string, DAE truncates at the end of the last complete symptom within the first 150 bytes of the symptom string.
- During its match processing, DAE performs a byte-by-byte comparison of symptom strings. Thus, be sure that the symptom strings you generate are consistent in the order of symptoms and the number of blank characters between symptoms.
- If the SYMREC parameter is used in a recovery environment where an SDWA exists, DAE uses the symptoms found in the symptom record rather than in the SDWA.

Requesting dumps of other systems

In a sysplex, a problem can involve several or all of the systems. The IEATDUMP macro can request a Transaction dump of the local system address space and, using the REMOTE parameter, a Transaction dump of one or more address spaces on the same or different systems. Using the parameters on the macro, you can make the dump requests the same or different, depending on the areas you think a diagnostician will need for any problems.

In systems in a sysplex, the installation should specify names that form patterns so that all the names can be specified in one value that contains wildcards. In the IEATDUMP macro, you can use wildcards, * and ?, to identify all or some names, based on the patterns for the names. For example, jobs can be named TRANS1, TRANS2, TRANS3, ... , TRANS17. A jobname of TRANS* in the IEATDUMP macro will dump all of them.

Note: If a Transaction dump uses the REMOTE parameter to dump one or more address spaces on a pre-release 4 system, the result will be a single SVC dump containing the requested data, instead of one or more Transaction dumps written to data set names specified with the DSN parameter. Issue the DISPLAY DUMP,STATUS command to determine the name of this SVC dump. In the

IEATDUMP macro, you can use wildcards in system names, address space identifiers, job names, XCF groups, XCF members, and data space names.

Chapter 21. Protecting the system

Protecting or maintaining system integrity is a major consideration in large systems. This topic includes information concerning the following:

- System integrity
- Using the authorized program facility (APF)
- Using the resource access control facility (RACF)
- System authorization facility (SAF)
- Using the command authorization service
- Changing system status (MODESET)
- Protecting the vector facility

System integrity

System integrity is defined as the inability of any program not authorized by a mechanism under the installation's control to circumvent or disable store or fetch protection, access a resource protected by the z/OS Security Server (RACF), or obtain control in an authorized state; that is, in supervisor state, with a protection key less than eight (8), or Authorized Program Facility (APF) authorized.

Documentation on system integrity

This topic contains information about system integrity. The related topic of security is discussed in the following publications:

- *The Considerations of Physical Security in a Computer Environment*
- *Data Security Controls and Procedures - A Philosophy for DP Installations*
- *Security Assessment Questionnaire*
- *MVS Planning: Security*

Installation responsibility

To ensure that system integrity is effective and to avoid compromising any of the integrity controls provided in the system, the installation must assume responsibility for the following:

- Physical environment of the computing system.
- Adoption of certain procedures (for example, the password protection of appropriate system data sets) that are a necessary complement to the integrity support within the operating system itself.
- That its own modifications and additions to the system do not introduce any integrity exposures. That is, all installation-written authorized code (for example, an installation SVC) must perform the same or an equivalent type of validity checking and control that the system uses to maintain its integrity.

Elimination of potential integrity exposures

System integrity support restricts only unauthorized problem programs. It is the responsibility of the installation to verify that any authorized programs added to the system control program will not introduce any integrity exposures. To do this effectively, an installation should consider these areas for potential integrity exposure:

- User-supplied addresses for user storage areas.
- User-supplied addresses for protected control blocks.
- Resource identification.
- SVC routines calling SVC routines.
- Control program and user data accessibility.
- Resource serialization. (See “Locking” on page 30.)

Each of the following descriptions is a guideline to aid the installation in:

- Eliminating the areas identified above as a potential integrity exposure.
- Determining whether an impact on existing installation-written code might occur, especially where that code is dependent on the use of non-intended interfaces to the system control program.

User-supplied addresses for user storage areas

A potential integrity exposure exists whenever a routine having PSW key 0-7 accepts a user-supplied address of an area to which a store or fetch is to be done. If the system routine does not adequately validate the user-supplied address to ensure that it is the address of an area accessible to the user for storing and fetching data, an integrity violation can occur when the system-key routine:

- Stores into (overlays) system code or data (for example, in the nucleus or the system queue area), or into another user's code or data.
- Moves data from a fetch-protected area that is not accessible to the user (for example, fetch-protected portion of the common service areas) to an area that is accessible to the user.

To eliminate this problem system-key routines should always verify that the entire area to be stored into, or fetched from, is accessible (for storing or fetching) to the user in question. The primary validation technique is the generally established convention that PSW key 0-7 routines obtain the protection key of the user before accessing the user-specified area of storage.

User-supplied addresses for protected control blocks

A potential integrity exposure exists whenever the control program (system key/privileged mode) accepts the address of a protected system control block from the user. For most system control blocks, this situation should not be permitted to exist. However, in certain cases it is necessary to allow the user to provide the address of a system control block that describes his allocation/access to a particular resource (for example, a data set), in order to identify that resource from a group of similar resources (for example, an user might have many data sets allocated). Inadequate validity checking in this situation can create an integrity exposure, because an unauthorized problem program could provide its own (counterfeit) control block in place of the system block and thereby gain the ability to:

- Access a resource in an uncontrolled manner (because the control block in this case would normally define the restrictions, such as read-only for a data set, on the user's allocation to the resource).
- Gain control in an authorized state (because such control blocks might contain the addresses of routines that run in supervisor state with PSW key 0-7.
- Cause various other problems depending on exactly what data is in the control block involved.

To avoid this type of exposure, the control program must verify, for every such address accepted from a problem program, that the address is that of:

1. A protected control block created by the control program.
2. The correct type of control program block (for example, a TCB versus a DEB, or a QSAM DEB versus an ISAM DEB).
3. A control block created for use in connection with the user (job step) that supplied the address.

In the system, verification is generally accomplished by establishing a chain or table of the particular type of control block to be validated. This chain or table is located via a protected and jobstep-related control block that is known to be valid. Addresses that are not allowed to be supplied by the user, are located via a chain of protected control blocks that begins with a control block known to be valid or fixed at a known location at IPL time, such as the CVT. Therefore, a control block can only be entered in the chain/table by:

- An authorized program satisfying point 1.
- Definition, where the chain/table establishes the type of control block satisfying point 2.
- Definition, where each chain/table is located only through a jobstep-related control block satisfying point 3.

Note: This does not imply that a system routine must go back to the CVT or similar control block every time it wants to establish a valid chain. Typically, a control block address not too far down on such a chain is available and already validated in a register.

Resource identification

Resource identification is another area that can be subject to integrity exposures. Exposures can result if the control program does not maintain and use sufficient data to uniquely distinguish one resource from other similar resources. For example, a program must be identified by both name and library to distinguish it from other programs. The consequences of inadequate resource identification are problems such as the ability of an unauthorized problem program to create counterfeit control program code or data, or to cause varying types of integrity problems by intermixing incompatible pieces of control program code or data, or both.

The general solution can only be stated as the reverse of the problem; that is, the control program must maintain and use sufficient (protected) data on any control program resource to distinguish between that resource and other control program or user resources. The following are examples of the controls that the system employs to comply with the requirement:

- In general, authorized program requests to load other authorized programs are satisfied only from authorized system libraries (see “Control Program Extensions” described in this information.)
- The operating system takes explicit steps to ensure that routines loaded from authorized system libraries are used only for their intended purpose. This includes expanded validity checking to remove any potential for the unauthorized program to specify explicitly which of the authorized library routines are to gain control in any given situation.
- Sensitive system control blocks are validated as being the “correct” blocks to be used in any given control program operation. (See “User-Supplied Addresses of Protected Control Blocks” described earlier in this information.)

SVC routines calling SVC routines

A potential problem area exists whenever a problem program is allowed to use one SVC routine (routine A) to invoke a second SVC routine (routine B) that the problem program could have invoked directly. An integrity exposure occurs if:

- SVC routine B bypasses some or all validity checking based on the fact that it was called by SVC routine A (an authorized program) or
- User-supplied data passed to routine B by routine A either is not validity checked by routine A, or is exposed to user modification after it was validated by routine A.

These problems will not exist if the user calls SVC routine B directly, because the validity checking will be performed on the basis of the caller being an unauthorized program.

SVC routine A, which is aware that it has been called by an unauthorized program, must ensure that the proper validity checking is accomplished. However, it is usually not practical for SVC routine A to do the validity checking itself, because of the potential for user modification of the data before or during its use by SVC routine B. The general solution should be for SVC routine A to provide an interface to SVC routine B, informing routine B that the operation is being requested with user-supplied data in behalf of an unauthorized problem program (implying that normal validity checking should be performed).

In practice, most of the SVC B-type system routines that could be subject to this problem use the key of their caller as a basis for determining whether or not to perform validity checking. Therefore, most SVC A-type system routines have simply adopted the convention of assuming the key of their caller before calling the SVC B routine. (For additional information, see “Writing SVC Routines”.)

Control program and user data accessibility

Important in maintaining system integrity is the consideration of what system data is sensitive and must be protected from the user, and what data can be exposed to user manipulation. The implications of the exposure of the wrong type of data are obvious.

In general, it is necessary to store protect the following types of data:

- Code, and the location of code, that is to receive control in an authorized state.
- Work areas for such code, including areas where it saves the contents of registers.
- Control blocks that represent the allocation or use of system resources.

The operating system maintains such items in its storage, or in a separate address space in the case of some APF-authorized programs.

It might also be necessary to protect, for a limited period, certain data that is normally under the control of the user (for example, to prevent its modification during a critical operation). In this case the system provides fetch protection for such data if:

- The data consists of proprietary information (such as passwords).
- The control program cannot determine the nature of the contents of the data area.

Fetch protection provided for the PSA

The last 2K locations of the PSA (addresses 2048 through 4095) contain sensitive system data that must be protected. These locations are key 0 fetch protected. This means that only key 0 programs can fetch data from the last 2K of the PSA. Also the entire PSA of one processor is key 0 fetch-protected from programs attempting to access the PSA while executing on another processor.

Control program extensions

This potential problem area involves the somewhat hazy distinction that exists between the control program and certain types of unauthorized programs. In most installations, there are unauthorized programs that are actually extensions to the control program in that they are allowed (by means of various special SVCs, and so forth) to bypass normal system controls over access to system resources. For example, a special utility program that scans all the data on a pack might be able to avoid the normal system extent checking on a direct access volume.

If an installation has its own control program extensions and SVCs that allow the bypass of normal system security or integrity checks (for example, an SVC that returns control in key 0), and if such SVCs are not currently restricted from use by an unauthorized program, the APF should be used to restrict them and to authorize the control program extensions that use them.

Authorized programs

Many system functions, such as entire supervisor calls (SVC) or special paths through SVCs, are sensitive. Access to these functions must be restricted to authorized programs to avoid compromising the security and integrity of the system. A program must be authorized before it can access a restricted SVC. The system considers a program *authorized* if the program has one or more of the following characteristics:

- Runs in supervisor state (bit 15 of the PSW is zero)
- Runs with PSW key 0-7 (bits 8-11 of the PSW are in the range 0-7)
- Runs under an APF-authorized job step task.

The system does not allow APF-authorized programs to use some resources that programs running in supervisor state or PSW key 0-7 are allowed to use. For example, certain macro keywords are restricted to programs running in supervisor state or PSW key 0-7. Programs that are APF-authorized, but not running in supervisor state or with PSW key 0-7, cannot use these keywords when invoking the associated macros.

MVS itself has authorized programs, which are:

- Programs residing in SYS1.LINKLIB or SYS1.SVCLIB
- SVC routines
- Program call (PC) routines
- Certain exit and I/O appendage routines that are called by authorized programs.

Any user can submit a job that runs an authorized program. To restrict a program to an individual user or a group of users, you can use library security facilities to place the program in a library (other than SYS1.LINKLIB, SYS1.SVCLIB, or a library in the LPALST) that is protected by a security product such as RACF. If a program is an APF-authorized program, it must reside in a library that is in the APF list or in the link pack area (pageable LPA, modified LPA, fixed LPA, or dynamic LPA).

Note: You can also restrict the use of a program by defining it in the RACF PROGRAM class, unless the program resides in the link pack area. See *z/OS Security Server RACF Security Administrator's Guide* for more information about functions of program control.

Using APF to restrict access to system functions

The authorized program facility (APF) allows your installation to identify system or user programs that can use sensitive system functions.

APF:

- Restricts the use of sensitive system SVC routines (and sensitive user SVC routines, if you need them) to APF-authorized programs
- Allows the system to fetch all modules in an authorized job step task only from authorized libraries, to prevent programs from counterfeiting a module in the module flow of an authorized job step task.

To authorize a program, the installation must first assign the authorization code to the first load module of the program. APF prevents authorized programs from accessing any load module that is not in an authorized library. When the system attaches the first load module of a program, the system considers the program *APF-authorized* if the module meets *both* of the following criteria:

- The module is contained in an authorized library or resides in the link pack area (pageable LPA, modified LPA, fixed LPA, or dynamic LPA) (see "APF-authorized libraries" on page 447).
- The module is link-edited with authorization code AC=1 (to indicate that you want to authorize the job step task). This code is contained in a bit setting in the partitioned data set (PDS) directory entry for the module. For more information about how to assign an authorization code to a module, see "Assigning APF authorization to a load module" on page 450.

If the system does not consider a program *APF-authorized* when it attaches the first load module, the program cannot become authorized for the life of the job step.

Note: This description applies to batch jobs and started tasks, where the initiator attaches the jobstep task and determines the APF authorization for that jobstep. The TSO/E terminal monitor program (TMP), and UNIX System Services can also run programs with APF authorization, as the initiator does. Other system environments generally do not support running programs with APF authorization.

The *authorization code* (AC) is meaningful only when the load module resides in an authorized library and runs as the first module of a job step task, or when run by the TSO/E terminal monitor program or UNIX System Services with appropriate configuration parameters. When a program is run with APF authorization, the system verifies that all subsequent modules for that program are contained in authorized libraries or the link pack area (pageable LPA, modified LPA, fixed LPA, or dynamic LPA). If one or more of the programs are not contained in authorized libraries or the link pack area, the system issues abend X'306'.

Guidelines for using APF

Installations using APF authorization must control which programs are stored in authorized libraries and in the link pack area (pageable LPA, modified LPA, fixed LPA, and dynamic LPA). If the first module in a program sequence is authorized, the system assumes that the flow of control to all subsequent modules is known and secure as long as these subsequent modules come from authorized libraries or the link pack area. To ensure that this assumption is valid, the installation should:

- Ensure that all programs that run as authorized programs adhere to the installation's integrity guidelines.
- Ensure that no two load modules with the same name exist across the set of authorized libraries or the link pack area. Two modules with the same name could lead to accidental or deliberate mix-up in module flow, possibly introducing an integrity exposure.
- Link edit with the authorization code (AC=1) only the first load module in a program sequence. Do not use the authorization code for subsequent load modules, thus ensuring that a user cannot call modules out of sequence, or bypass validity checking or critical logic flow.

IBM recommends that you protect the libraries in the APF list with a security product, such as RACF, and ensure that only appropriate users with system maintenance responsibilities can update these libraries. You should also apply similar controls to any library that contributes modules to the link pack area (pageable LPA, modified LPA, fixed LPA, or dynamic LPA) and to any libraries specified in RACF PROGRAM profiles.

APF-authorized libraries

APF-authorized programs must reside in the link pack area (pageable LPA, modified LPA, fixed LPA, dynamic LPA) or in an authorized library:

- SYS1.LINKLIB
- SYS1.SVCLIB
- Another library in the linklist (depending on the LNKAUTH= parameter in your PARMLIB members)
- Another authorized library specified by your installation.

The LNKLSTxx parmlib member indicates the libraries that are to be concatenated to SYS1.LINKLIB. The libraries in the LNKLST concatenation are considered authorized unless the system programmer specifies LNKAUTH=APFTAB in the IEASYSxx parameter list. If the system accesses the libraries in the LNKLST concatenation through JOBLIB or STEPLIB DD statements, the system does not consider those libraries authorized unless you enter the library names in the APF list using one of the methods described in “APF-authorized library list” on page 448. For more information about the LNKLSTxx parmlib member, see *z/OS MVS Initialization and Tuning Reference*.

If a load module resides in the link pack area or in an authorized library, an authorized program can load the module. To help avoid integrity exposures, do not duplicate module names across the link pack area or the authorized libraries.

Note:

1. If a JCL DD statement concatenates an authorized library in any order with an unauthorized library, the entire set of concatenated libraries is treated as *unauthorized*.
2. SYS1.LPALIB and other libraries that contribute to the link pack area (pageable LPA, modified LPA, fixed LPA, dynamic LPA) are treated as authorized when the system places modules into the link pack area. You should protect those libraries as you would protect any APF-authorized library. When accessed via a tasklib DCB, or via a STEPLIB or JOBLIB DD statement, these libraries are considered authorized only if you have specified them in the APF list.

APF-authorized library list

The libraries that contain APF-authorized programs must be defined in an APF list. The APF list contains the following information for each library:

- The library name
- An identifier for the volume that contains the library (or “SMS”, if the library is SMS-managed).

Note: The system considers all modules in the link pack area (pageable LPA, modified LPA, fixed LPA, dynamic LPA) as having come from an authorized library whether or not the library is defined in the APF list).

The system automatically places SYS1.LINKLIB and SYS1.SVCLIB in the first two APF list entries. Your installation can create the remaining entries in the APF list.

Note: When LNKAUTH=APFTAB is specified, the system considers SYS1.MIGLIB, SYS1.CSSLIB,SYS1.SIEALNKE and SYS1.SIEAMIGE to be APF-authorized when they are accessed as part of the concatenation (even when they are not included in the APF list).

Defining Aliases in the APF List: Do not define aliases in the APF list because data management services of IBM (for example, OPEN processing) map an alias to its actual library name and query the APF list by the actual library name. An alias in the APF list thus does not actually authorize anything.

You can specify a dynamic format for the APF list, which allows you to:

- Update the APF list without having to reIPL the system, and
- Specify as many APF-authorized libraries as you need; there is no system-imposed maximum number.

You can also specify a static format for the APF list, which has the following limitations:

- You can define the static APF list only at IPL, using the IEAAPFxx or PROGxx parmlib member
- You can only specify a maximum of only 253 libraries in the static APF list (255 entries are allowed, but SYS1.LINKLIB and SYS1.SVCLIB occupy two entries in the list).

Because of the limitations associated with a static APF list, IBM recommends that you maintain a dynamic APF list.

“Requesting APF list services (CSVAPF macro)”explains how to create, update, and change the format and contents of the APF list.

Requesting APF list services (CSVAPF macro)

As you read this information, use the description of the CSVAPF macro in *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

Programs can issue the CSVAPF macro to:

- Change the format of the APF list from static to dynamic.
- Add and delete libraries in a dynamic APF list (you cannot use CSVAPF to add and delete entries in a static APF list).

Your installation can also use a parmlib member or an operator command to specify the format and contents of the APF list. For more information, see the descriptions of the PROGxx and IEAAPFxx parmlib members in *z/OS MVS Initialization and Tuning Reference* and the descriptions of the SETPROG and SET PROG=xx commands in *z/OS MVS System Commands*.

Before you change the format of the APF list to dynamic, contact the system programmer to validate that all programs and vendor products are converted to use dynamic APF services and that the proper program products are installed.

Restricting the use of SVC routines

You can use one of the following methods to restrict the use of sensitive SVC routines to authorized callers:

1. Specify the APF=YES parameter on the SVCUPDTE macro, or specify APF(YES) in the IEASVCxx parmlib member. With this specification in place, a X'047' abend results if an unauthorized program tries to access the restricted SVC.
2. Insert the TESTAUTH macro at appropriate locations in an SVC routine to return the caller's authorization. Then the SVC routine can restrict either the entire routine or particular paths through the routine to authorized callers.

   When using TESTAUTH to return the caller's authorization, test only the criteria (supervisor state, PSW key 0-7, APF authorized or any combination of the three) that you consider essential. TESTAUTH returns an indication of whether the calling program meets any of the three criteria. For example, various system functions use TESTAUTH to make the following distinctions:

   a. The caller is running in supervisor state, PSW key 0-7, or both.
   b. The caller is an APF-authorized task.
   c. Either a or b.

The TESTAUTH macro is not used to test the use of I/O appendages, which are instead controlled through the IEAAPP00 parmlib member. (See the description of this member in *z/OS MVS Initialization and Tuning Reference*.)

Restricting load module access

To authorize a program, the installation must:

1. Assign the authorization code AC(1) to the first load module of the program; and
2. Place the module in an APF-authorized library or in the link pack area; and
3. Invoke the program via the initiator (// EXEC PGM=program in batch or an STC), or via the TSO/E terminal monitor program (with appropriate configuration statements in SYS1.PARMLIB(IKJTSOxx), or via UNIX System Services (again, possibly requiring configuration statements in SYS1.PARMLIB(BPXPRMxx).

When the program runs with APF authorization, the system prevents it from accessing any other load module that is not in either the link pack area or an authorized data set. If an authorized program tries to access a module that is not in an authorized data set, the system searches the authorized data sets for a copy of the module. If the system finds a copy of the module, processing continues. If the system does not find a copy of the module, the system issues abend X'306'.

Assigning APF authorization to a load module

You can use the PARM field on the link edit step to assign the APF-authorization code to a load module. To assign an authorization code using JCL, code AC=1 in the operand field of the PARM parameter of the EXEC statement:

```
//LKED   EXEC  PGM=HEWL,PARM='AC=1',...
```

This method causes the system to consider every load module created by the linkage editor to be *authorized*.

The authorization code of a load module has meaning only when it resides in the link pack area or an APF-authorized data set and when the initiator attaches it as a jobstep task in a batch job or STC, or when the TSO/E terminal monitor program attaches the program (with appropriate TSO/E configuration) or when UNIX System Services attaches the program (again with appropriate UNIX System Services configuration). If no authorization code is assigned in the linkage editor step, the system assigns the default of *unauthorized*.

No security or integrity exposure exists if a program is link-edited into an unauthorized data set with authorization code AC=1. The job step task is not authorized when the first module of the job step task is loaded and no abend occurs. However, if the loaded module tries to execute functions or SVCs that require authorization, the system abnormally ends the program.

Overriding an authorization code - SETCODE statement

You can override the authorization code for an output load module with the SETCODE control statement. The SETCODE statement establishes authorization for a specific output load module. You must place the SETCODE statement before the NAME statement for the load module. The format of the SETCODE statement is:

```
SETCODE   AC(1)
```

If you assign more than one SETCODE statement to a given output load module, the system uses the last SETCODE statement coded.

Figure 62 shows how the SETCODE statement assigns an authorization code to the output load module MOD1:

```
//LKED      EXEC  PGM=HEWL
//SYSPRINT  DD    SYSOUT=A
//SYSUT1    DD    UNIT=SYSDA,SPACE=(TRK,(10,5))
//SYSLMOD   DD    DSNAME=SYS1.LINKLIB,DISP=OLD
//SYSLIN    DD    DSNAME=&&LOADSET,DISP=(OLD,PASS),
//                UNIT=SYSDA
//          DD    *
  SETCODE         AC(1)
  NAME            MOD1(R)
/*
```

Figure 62. Assigning Authorization via SETCODE

Authorization results under various conditions

When a program issues an SVC or accesses a load module through a LINK, LOAD, or XCTL macro, the system considers the following factors when checking authorization:

- Whether or not the calling program is authorized
- Whether the called program is a restricted SVC or a load module in an authorized data set or the link pack area.

Table 64 describes the authorization rules:

Table 64. Authorization Rules

Rule	Abend Resulting From Violation
An unauthorized routine cannot call a restricted SVC.	047
A routine running in supervisor state, system key, or APF-authorized cannot call programs residing outside APF-authorized data set or the link pack area.	306

The rules shown in Table 64 are also true when the ATTACH macro is used unless the RSAPF keyword is specified. An attaching task that specifies RSAPF=YES and is running in supervisor state or PSW key 0-7 can attach programs residing outside APF-authorized data set or the link pack area if the following conditions are met:

- The caller is not running APF-authorized.
- The caller is attaching a subtask in problem state.
- The attached task's TCB key is 8-15 (non-system key).

The newly attached subtask does not run APF-authorized. If the attaching task is not in supervisor state or PSW key 0-7, the default, RSAPF=NO, is taken and a 306 abend might result.

However, if the subtask comes from an APF-authorized data set or the link pack area and is link edited with the APF-authorized attribute, then the task executes with APF authorization.

Resource Access Control Facility (RACF)

The Resource Access Control Facility (RACF) provides software access control measures that can be used to enhance data security in a computing system. RACF can be used in addition to any data security measure currently being used.

RACF provides the ability to specify access authorities under which the permanent DASD data sets, tape volumes, DASD volumes, terminals, and other resources are made available to the users of the system. RACF can protect VSAM, non-VSAM, cataloged, and uncataloged data sets, including program libraries and individual programs.

When the security administrator defines users, groups, DASD data sets, tape volumes, DASD volumes, terminals, and other resources to RACF, RACF builds and stores their descriptions in profiles on the RACF data base. RACF uses these profiles for authorization checking and user identification and verification.

For a description of the RACROUTE macro, see *z/OS Security Server RACROUTE Macro Reference* For information on how to use RACF to protect resources, see *z/OS Security Server RACF Security Administrator's Guide*.

System Authorization Facility (SAF)

The System Authorization Facility (SAF) provides a system interface that conditionally directs control to the Resource Access Control Facility (RACF), if RACF is present, and/or a user-supplied processing routine when receiving a request from a resource manager. Another external security product may be substituted for RACF. SAF does not require any other program product as a prerequisite, but overall system security functions are greatly enhanced and complemented by the concurrent use of an external security product. The key element in SAF is the MVS router.

MVS router

SAF provides an installation with centralized control over system security processing by using a system service called the MVS router. The MVS router provides a focal point and a common system interface for all products providing resource control. The resource managing components and subsystems call the MVS router as part of certain decision-making functions in their processing, such as access control checking and authorization-related checking. These functions are called “control points”. This single SAF interface encourages the use of common control functions shared across products and across systems.

The router is always present whether or not an external security product is present. If an external security product is available in the system, the router passes control to the external security product. Before it calls the external security product, the router calls an optional, user-supplied security processing exit if one has been installed.

Control points that issue the RACROUTE macro enter the MVS router in the same key and state as the RACROUTE issuer. Control points that continue to issue the independent RACF system macros (RACDEF, RACINIT, RACHECK, RACLIST, RACXTRT, and FRACHECK) go directly to the external security product, bypassing the router.

MVS router exit

The MVS router provides an optional installation exit that is invoked whether or not RACF or another external security product is installed and active on the system. If an external security product is not available, the router exit acts as an installation-written security processing (or routing) routine. If an external security product is available, the exit acts as an external security product preprocessing exit.

For more information on the MVS router exit, see *z/OS MVS Installation Exits*.

Interface to the MVS router (RACROUTE)

The RACROUTE macro accepts all valid parameters for any of the independent RACF system macros (RACDEF, RACINIT, RACHECK, RACLIST, RACXTRT, and FRACHECK) and internally issues the appropriate independent RACF system macro to generate a RACF parameter list. When the RACROUTE macro internally invokes the independent RACF system macros, RACROUTE verifies that only valid parameters have been coded and then passes the parameters to the MVS router.

Existing control points that invoke external security product processing via the supervisor call interface can continue to do so or can replace the supervisor calls with the RACROUTE macro.

Note: To invoke RACF or RACF-compatible functions that were available in RACF 1.8 or an earlier release, you can use the independent RACF system macros directly or you can use the RACROUTE macro. IBM recommends that you use the RACROUTE macro. To use the new RACF or RACF-compatible functions, you **must** use the RACROUTE macro.

See *z/OS Security Server RACROUTE Macro Reference* for a description of the RACROUTE macro and the RACROUTE return and reason codes.

Defining a resource (RACROUTE REQUEST=DEFINE)

The RACROUTE macro with the REQUEST=DEFINE parameter defines or deletes a resource profile (for example, a tape volume profile or a DASD data set profile) for an external security product.

Identifying a user (RACROUTE REQUEST=VERIFY and REQUEST=VERIFYX)

RACROUTE REQUEST=VERIFY: The RACROUTE macro with the REQUEST=VERIFY parameter determines if a userid is defined to an external security product and if the user has supplied a valid password, group name, submitter node, security label, or operator identification. The external security product builds an access environment element for the user if the userid, password, group name, and terminal id (for the terminal user) are accepted. The identification and verification in the case of a terminal or batch job user, is based on the information contained in the TSO LOGON or IMS™ /SIGN command or data specified in the JOB statement for the batch job. The access environment element identifies the scope of the user's authorization to be used during the current terminal session or batch job.

RACROUTE REQUEST=VERIFYX: The RACROUTE macro with the REQUEST=VERIFYX parameter uses information about the caller to generate the correct security identity of the new job. It verifies that a submitted job has a valid user ID, group ID, security label, and password. The verified job information fills a UTOKEN (a security token associated with a user) that RACROUTE passes to the caller.

Checking resource authorization (RACROUTE REQUEST=AUTH and REQUEST=FASTAUTH)

RACROUTE REQUEST=AUTH: The RACROUTE macro with the REQUEST=AUTH parameter determines if a user is authorized to obtain use of a resource (for example, DASD data set, tape volume, or DASD volume) protected by the external security product. When a user requests access to a protected resource, acceptance of the request is based upon the identity of the user and whether the user has been permitted sufficient access authority to the resource.

The external security product performs system authorization checking when a resource manager that controls a protected resource issues RACROUTE REQUEST=AUTH before allowing a user access to the resource.

The system programmer using this macro to check a user's authorization to a resource has available three parameters (CSA, LOG, and PROFILE) that are not available to the application programmer. These parameters permit the system programmer to specify that a profile is to be copied and maintained in central storage for the resource and that different types of access attempts are or are not to be recorded on the SMF data set.

RACROUTE REQUEST=FASTAUTH: The RACROUTE macro with the REQUEST=FASTAUTH parameter provides a fast-path way to perform a function similar to RACROUTE REQUEST=AUTH. RACROUTE REQUEST=FASTAUTH, however, requires that the profile of the resource being checked be in storage. To build an in-storage profile, issue RACROUTE REQUEST=LIST before issuing RACROUTE REQUEST=FASTAUTH.

Retrieving and encoding data (RACROUTE REQUEST=EXTRACT)

The RACROUTE macro with the REQUEST=EXTRACT parameter can be used to retrieve or update fields from any profile maintained by RACF or an external security product. RACROUTE REQUEST=EXTRACT can also be used to encode certain clear-text (readable) data.

Building in-storage profiles (RACROUTE REQUEST=LIST)

The RACROUTE macro with the REQUEST=LIST parameter builds in-storage profiles from class resources defined to the external security product. RACROUTE REQUEST=LIST processes only general resources. Once RACROUTE REQUEST=LIST brings profiles into central storage, you can issue RACROUTE REQUEST=AUTH for the resources and the in-storage profiles are used for authorization checking.

Checking auditing options (RACROUTE REQUEST=AUDIT)

The RACROUTE macro with the REQUEST=AUDIT parameter checks the auditing options for a given resource and class, and logs a record for a given event.

Checking user authority (RACROUTE REQUEST=DIRAUTH)

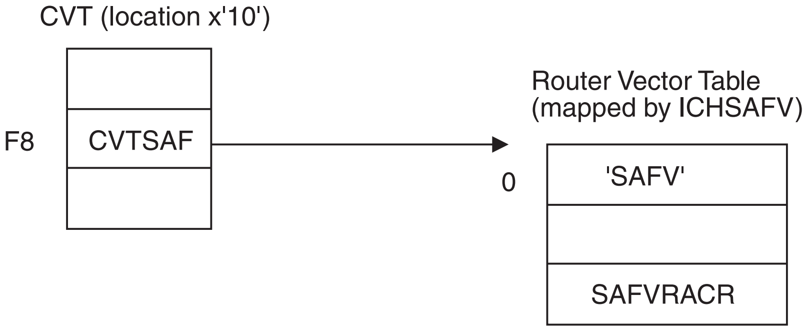
The RACROUTE macro with the REQUEST=DIRAUTH parameter verifies the mandatory access authority of the user to a resource. It allows the system to audit the control of message transmission at the resource class level.

SAF interface to an external security product

If RACF is not present in the system, an installation can use an external (non-RACF) security product to provide system security functions. An external security product supplies module ICHRFR00 (called the security product router) as the entry for security processing from SAF.

The MVS router passes control to ICHRFR00 when a control point issues the RACROUTE macro to request a security function that is not completely processed by SAF.

In addition to supplying module ICHRFR00, the external security product must also place the address of ICHRFR00 into field SAFVRACR in the router vector table (mapped by ICHSAFV). The router vector table is built during SAF initialization, but the field SAFVRACR should be filled during the initialization of the external product. The following illustration describes the location of the field SAFVRACR.



Requirements for the external security product router

Module ICHRFR00 is an LPA-resident module. Installation procedures for the security product put ICHRFR00 in SYS1.LPALIB. If the installation replaces ICHRFR00, a re-IPL is needed for the change to take effect.

The following general requirements apply for module ICHRFR00.

Environmental factor	Requirement
Authorization:	Supervisor state or problem state, and in any PSW key
Dispatchable unit mode:	Task mode or SRB mode
Cross-memory mode:	PASN = HASN = SASN
Amode:	24-bit or 31-bit (same as caller of RACROUTE request)
ASC mode:	Primary
Locks:	No locks held
Linkage conventions:	Standard

Input parameters to the external security product router

On entry to ICHRFR00, register 1 contains the address of the following area:

Table 65. Input Parameters to the External Security Product Router

Offset	Length	Description
0	4	**Paramater list address** - points to the RACROUTE parameter list (mapped by macro ICHSAFP). See data area SAFP in *z/OS Security Server RACROUTE Macro Reference* and *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).
4	4	**Work area address** - points to a 152-byte work area that the module can use.

For more information, see *z/OS Security Server RACROUTE Macro Reference* and see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Return and reason codes from the external security product router

On exit from module ICHRFR00 and return to the MVS router:

- Register 15 contains one of the following return codes:

Hex (Decimal)	**Meaning**
0 (0)	The requested function completed successfully.
4 (4)	The requested function was not processed.
8 (8)	The requested function was processed and has been failed.

- The RACROUTE parameter list (SAFP) contains the function return code in field SAFPRRET and the function reason code in field SAFPRREA for the requested function.

  See *z/OS Security Server RACROUTE Macro Reference* for a description of the RACROUTE function return and reason codes.

Programming considerations

When an external security product processes a RACROUTE REQUEST=VERIFY request, SAF expects the product to build data area ACEE, which is mapped by macro IHAACEE. For token processing, SAF uses field ACEETOKP to reference the token. Token processing is not required, but to be fully compatible with MVS, the security product should fill in field ACEETOKP with the address of the token.

SAF does not reference any other areas that are chained off the ACEE; therefore, the security product need not fill in the addresses of any other data areas in the ACEE.

Using the command authorization service

Command processors that need to find out whether a user is RACF-authorized to issue a command, should use the command authorization service. If you are not sure whether your command processor should use this service, check with your installation security administrator. Use of the command authorization service increases the path length required to process a command.

To use the command authorization service, a command processor issues the CMDAUTH macro. The service provides a return code that indicates the user's authorization status. Under certain conditions, the command authorization service is unable to make a decision. For example, a decision cannot be made if RACF 1.9 or later is not installed or is inactive. A return code and reason code together indicate why a decision could not be made. IBM recommends the use of this service by all command processors that must verify the authorization status of a user.

The CMDAUTH macro has a number of keywords that are functionally equivalent to corresponding RACROUTE macro keywords. These keywords have the same characteristics as their corresponding RACF macro keywords. The following discussion identifies these keywords. The CMDAUTH macro is described in *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*. The RACF macros are described in *z/OS Security Server RACF Macros and Interfaces*.

The CMDAUTH macro has several required keywords. To indicate whether you want the list form or the execute form of the macro, use the MF keyword. You must also supply the resource name of the command being checked and indicate the access authority for which the service is to check. For an explanation of how to create a command resource name, see "Command resource names" on page 457. To provide the resource name, use the ENTITY keyword. To indicate the access authority, use the ATTR keyword. The ENTITY keyword and the ATTR keyword correspond to similar keywords on the REQUEST=AUTH form of the RACROUTE macro.

In addition, the CMDAUTH macro requires one of the following:

- The command text that is to be authority checked and the user security token that is to be used for the authority check. The LOGSTR keyword identifies the information that is to be included in the RACF SMF PROCESS records and the UTOKEN keyword identifies the user security token. These keywords correspond to similar keywords on the REQUEST=AUTH form of the RACROUTE macro. If you need to obtain a user security token, see *z/OS Security Server RACF Macros and Interfaces* for information on how to use the REQUEST=TOKENXTR form of the RACROUTE macro.
- A command input buffer (CIB), and an indication that you have provided a CIB. The CNTLBLK keyword identifies the CIB and the CBLKTYPE keyword indicates that you have provided a CIB.

The CMDAUTH macro also has several optional keywords. Each of these keywords correspond to similar keywords on the RACROUTE macro.

- The keywords MSGRTN, MSGSUPP, and MSGSP enable you to control the routing of messages related to your request. To indicate whether you want messages returned to your program, use the MSGRTN keyword. The default is to not return messages. To indicate whether you want to suppress messages issued by SAF and RACF, use the MSGSUPP keyword. If you do not want the operator to see the messages, suppress them. The default is to not suppress these messages. If messages are returned to your program, by default the system returns them to subpool 229. If you want the messages returned to a different subpool, use the MSGSP keyword to identify the subpool.

- To provide a control point name to RACF, use the REQSTOR keyword. To find out which control point names you can use, see your RACF administrator.
- To provide RACF with the name of the calling subsystem, use the SUBSYS keyword. To find out which subsystem names you can use, see your RACF administrator.

Command resource names

Command resource names (sometimes called entity names) enable you to logically group your own commands and name each group for the purpose of controlling access to the commands. After you have defined the groups of commands and given each group a resource name, your RACF administrator must define the command groups to RACF. You must also define who is allowed to issue commands from each group and have the RACF administrator supply this information to RACF. After the administrator has provided the required information to RACF, you can use the command resource name on the CMDAUTH macro to perform authorization checking.

Syntax

IBM recommends that command resource names use the syntax described in this topic.

A command resource name can include up to four parts: a system identifier, the command or a variation of the command, a command qualifier, and a command object. These parts enable you to define a naming hierarchy that can identify specific commands or command subsets.

All command resource names must include a system identifier and the command name or a variation of the command name. The use of the command qualifier and the command object is optional and depends on the naming structure you want to define.

Periods must separate the parts of a command resource name. A command resource name can be up to 39 bytes long including the periods.

```
system_identifier .command [.command_qualifier [.command_object]]
```

system_identifier

Identifies the system, subsystem, or application to which the command belongs. For example, IBM uses **MVS** to identify MVS operator commands, **JES2** to identify JES2 commands, and **JES3** to identify JES3 commands.

system_identifier is a required part of the name.

command

Identifies a specific command or some variation of a command. Where possible, use the command name. In cases where the command name does not provide the level of identification you require, use a variation of the command name. You might use a variation of a command name under the following conditions:

- When coding a particular keyword on a command causes command processing to be significantly different than if you omitted the keyword
- When coding a particular keyword on a command requires the issuer of the command to have a higher authority than is required if the keyword is omitted.

For example, the appearance of the ARM keyword on the MVS FORCE command causes significantly different processing than occurs when the ARM keyword is omitted. Therefore, if you were defining command resource names for the FORCE command, you might use the following variations:

MVS.FORCE.DEV.*device*
The resource name that identifies the FORCE *device* command.

MVS.FORCEARM.DEV.*device*
The resource name that identifies the FORCE *device*,ARM command.

command is a required part of the name.

[.*command_qualifier***]**
command_qualifier allows you to more precisely identify the command variation in question. For example, if you were defining a command resource name for the MVS SET command, you might use the keywords IOS or SMS. The command resource names would appear as:

- MVS.SET.IOS
- MVS.SET.SMS

command_qualifier is an optional part of the name.

[.*command_object***]**
command_object identifies the object of the command. For example, *command_object* might identify:

- the device on a CANCEL command
- the jobname on a MODIFY command

Including the *command_object* as part of the command resource name enables you to control access to commands based on the object the command affects.

command_object is an optional part of the name.

Examples

The following are examples of some MVS commands and their corresponding command resource names:

command
command resource name

CANCEL *jobname*
MVS.CANCEL.JOB.*jobname*

MVS.CANCEL.STC.*jobname*

CANCEL U=*userid*
MVS.CANCEL.TSU.*userid*

CANCEL *device*
MVS.CANCEL.DEV.*device*

SET CLOCK
MVS.SET.TIMEDATE

SET DATE
MVS.SET.TIMEDATE

For a list of the MVS provided command resource names, see *z/OS MVS Planning: Operations*.

Changing system status (MODESET)

The MODESET macro alters selective fields of the program status word (PSW) and other control information associated with the unit of work. It can update the PSW key, the PSW state bit, and the PSW Key Mask. You can code the standard form of MODESET in two separate ways: one form generates an SVC and the other form generates inline code.

Generating an SVC

This form of MODESET, which executes as APF-authorized, in supervisor state, or under PSW key 0-7, changes the status of programs between supervisor state and problem program state, and key zero and non-key zero. The parameters that must be specified to perform the changes are MODE and KEY respectively.

The MODE parameter specifies whether bit 15 of the PSW is to be set to one or zero. When PSW bit 15 is one, the processor is in the problem state. For problem state, the caller’s PSW Key Mask (PKM) is set according to the following rules:

- The bit matching the resulting PSW key is set on.
- The bit matching key 9 is set on.
- For a task attached with ATTACHX using the KEY=NINE parameter, the bits that were on in the PKM of the ATTACHX issuer are set on.
- All other bits are set off.

When PSW bit 15 is zero, the processor is in supervisor state and the PSW Key Mask is unchanged.

The KEY parameter specifies whether bits 8-11 are to be set to zero or set to the value in the caller's TCB. Bits 8-11 form the processor protection key. The key is matched against a key in storage whenever information is stored, or whenever information is fetched from a location that is protected against fetching.

Generating inline code

This form of MODESET is used to ensure that storage areas and the control program functions they are associated with have the same protection key. The EXTKEY parameter of MODESET indicates the key to be set in the current PSW.

You can set the following keys:

- Key of zero
- Key of TCB
- Key of caller of type 1 SVC issuing MODESET
- Key of caller of type 2, 3, or 4 SVC issuing MODESET

Other parameters of MODESET allow the original key to be saved and restored upon completion of the desired changes. The inline form of the MODESET macro does not result in a change to the caller’s PSW Key Mask.

Chapter 22. Exit routines

This information includes the following:
- Using asynchronous exit routines
- Establishing a timer disabled exit routine
- Using dynamic exits services.

Using asynchronous exit routines

An authorized caller can request an asynchronous exit routine to run on behalf of a specific task.

Before the asynchronous exit routine runs, the calling program must initialize an interrupt request block (IRB) to identify the exit to the system and then schedule the asynchronous exit to run. The system then completes processing to get the asynchronous exit ready to run. See "System processing to run an asynchronous exit" on page 466.

There are several macros you can use to initialize the IRB and schedule the exit routine:
- Initialize an IRB - The SCHEDIRB macro is the recommended interface for initializing an IRB for an asynchronous exit routine. See "Using the SCHEDIRB macro to initialize and schedule an IRB" on page 462.

  You can also use the CIRB macro to initialize IRBs. This method allows you to reuse IRBs. See "Using the CIRB macro to initialize an IRB" on page 463.
- Schedule the asynchronous exit - The SCHEDIRB macro is the IBM recommended interface for scheduling an asynchronous exit. Using SCHEDIRB both to initialize and to schedule the IRB allows you more control over when your exit runs.

  If you initialized the IRB for the exit using the CIRB macro, you can also use the SCHEDXIT macro to schedule your asynchronous exit, see "Using the SCHEDXIT macro to schedule an IRB" on page 465.

Asynchronous exits (IRBs) must save and restore all the Floating Point Registers and the Floating Point Control register if they use them. If an asynchronous exit uses Binary Floating Point (BFP) instructions, it can cause the IEEE exception flags and DXC in the Floating Point Control (FPC) register to change at unpredictable times relative to the main program. Therefore, asynchronous exits that use BFP instructions must save and restore the entire Floating Point Control register.

Note: Asynchronous exits that use decimal arithmetic instructions can cause the DXC to be set to zero if they take a data exception and the MVS task is enabled to save and restore the Additional Floating Point and Floating Point Control registers. In this case the asynchronous exit may, but is not required to, save and restore the Floating Point Control register. (The main program should not be affected by the DXC being set to zero because the DXC is normally used only when handling a data exception.)

Using the SCHEDIRB macro to initialize and schedule an IRB

The SCHEDIRB macro is the recommended interface for initializing an IRB for an asynchronous exit and scheduling the exit routine to run. Use the SCHEDIRB macro to initialize and schedule an exit routine, rather than the CIRB and SCHEDXIT macros.

When you use the SCHEDIRB macro both to initialize the IRB and to schedule the asynchronous exit to run, you can control when it runs by scheduling the IRB for the exit in the following ways:

- Schedule the IRB to run under any task in the current address space by specifying a TCB address on the TCBPTR parameter.
- Schedule the IRB to run prior to any RB under the current task in the current address space by specifying an RB address on the RBPTR parameter. This type of IRB is called a directed IRB.

The best way to use directed IRBs is to make sure that the calling program is running under an IRB. If the system has suppressed asynchronous exits or the current task is in process-must-complete mode when SCHEDIRB is invoked, the calling program will get a non-zero return code unless the calling program is running under an IRB. You can make sure the calling program is running under an IRB by first invoking the SCHEDIRB macro with the TCBPTR option or by invoking the STIMER macro.

Detailed information about coding the SCHEDIRB macro appears in *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

The options you choose to specify on the SCHEDIRB macro affect the characteristics of the asynchronous exit routine. Consider the following when coding the SCHEDIRB macro:

- The exit routine runs in the mode specified on the MODE parameter.
- The exit routine runs in the key specified on the KEY parameter.
- If you specify the TCBPTR parameter, you can also request a save area using the SVAREA parameter.
- SCHEDIRB passes a parameter list to the exit routine if you specify it on the PARAMPTR parameter.
- Register contents upon entry to the exit routine are:

Register	**Contents**
0	Does not contain any information for use by the exit routine
1	Parameter list address (PARAMPTR), if any
2-12	Do not contain any information for use by the exit routine
13	Problem program register save area address, if SVAREA=YES was specified.
14	Return address
15	Entry point address

Using the SCHEDIRB macro to schedule an IRB

You can also use the SCHEDIRB macro only to schedule an initialized IRB for an asynchronous exit. In this case, use the CIRB macro to initialize the IRB and SCHEDIRB to schedule the IRB initialized by the CIRB macro. This technique allows you to take advantage of the capacity for reusing IRBs that the CIRB macro allows. On the SCHEDIRB macro, specify the IQEPTR parameter, using the address of the IQE for the IRB initialized by the CIRB macro. When you use SCHEDIRB with the CIRB macro, you cannot use the TCBPTR or RBPTR parameters that SCHEDIRB provides to control when your exit runs.

Using the CIRB macro to initialize an IRB

You can use the CIRB macro to initialize an IRB for an asynchronous exit. You can reuse an initialized IRB by omitting STAB=DYN on the CIRB macro. Thus, you need to invoke the CIRB macro only once for two or more invocations of an exit routine.

Detailed information about using the CIRB macro appears in *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

When the system returns control to the caller, it returns the IRB address in register 1. The IQE, if requested via the WKAREA parameter of the CIRB macro, occupies storage contiguous to the IRB and is pointed to by the RBNEXAV field in the IRB. The problem program work area, if requested, is pointed to by the RBPPSAV1 word in the IRB. The data area configuration is shown in Figure 63.

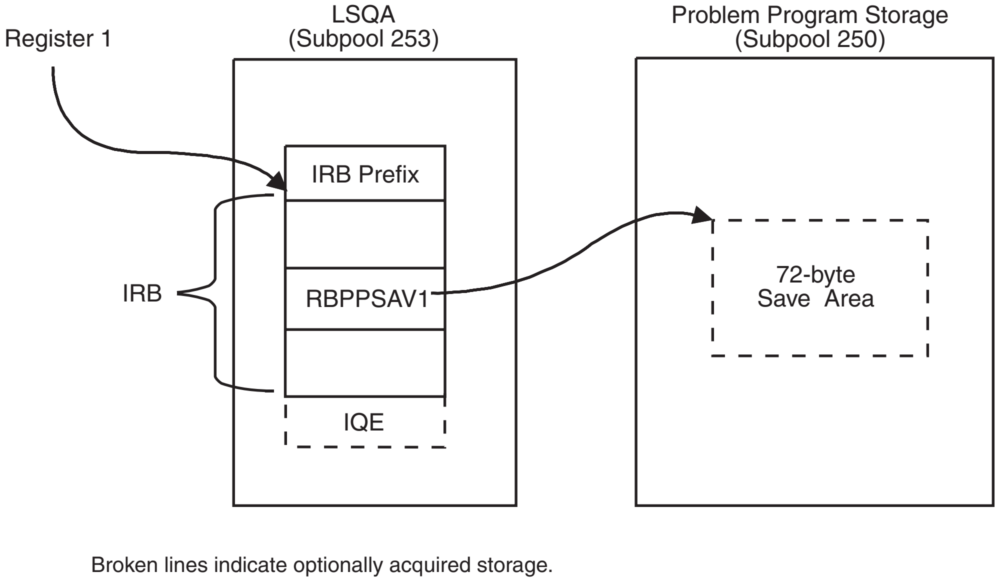


Figure 63. Asynchronous Exit Data Area Configuration

The caller must initialize the IQE. The fields to be initialized are:

- IQEPARAM -- optional address of the parameter list to be passed to the exit routine.
- IQEIRB -- address of the IRB as returned in register 1 by the CIRB macro.
- IQETCB -- address of the TCB for the task under which the user's exit routine is to run. If this is not the task that issued CIRB then, if you specified the STAB=DYN option of the CIRB macro to ask the system to free the IRB and IQE, you need to have specified the BRANCH=YES option of the CIRB macro, having placed the address of the IQETCB task into register 4 before issuing CIRB, in order for the freeing to succeed.

The fields in the IRB initialized by the CIRB macro are:

- RBEP -- entry point address of the exit routine.
- RBSTAB -- flags indicating how the IRB and IQE are to be treated upon termination of the exit routine (defined according to the STAB and RETRN parameters of CIRB).
- RBIQETP -- flag indicating the type of queue element (RQE or IQE) associated with the exit request.

  Note: Only the EXCP processor uses RQEs.
- RBSIZE -- the size of the IRB (including the size of the IQE if the CIRB specification included the WKAREA parameter).
- RBOPSW -- PSW to be loaded to initiate execution of the exit routine:
  - PSW is enabled for interrupts.
  - Protection key: 0 if KEY=SUPR specified on CIRB macro; TCB key of the caller if KEY=PP is specified on CIRB.
  - Mode: Supervisor state if MODE=SUPR on the CIRB macro; problem program state if MODE=PP on CIRB.
  - The AMODE parameter of the CIRB macro determines the addressing mode.
- RBNEXAV -- the address of the IQE if WKAREA was specified (occupies the first four bytes of the work area). This field is also used as the anchor of the queue of IQEs for completed IQEs which specified RETIQE=NO and RETRN=YES.
- RBPPSAV1 -- the address of the problem program save area if SVAREA was specified.

The options you choose to specify on the CIRB macro affect the characteristics of the asynchronous exit routine. Consider the following when coding the CIRB macro:

- The exit routine runs enabled in the key and state requested by the CIRB macro.
- Upon entry, the exit routine runs with an empty dispatchable unit access list (DU-AL). To establish addressability to a data space created by the mainline routine, the exit routine can use the ALESERV macro with the ADD parameter, and specify the STOKEN of the data space.
- Register contents upon entry to the exit routine are:

Register	**Contents**
0	IQE address
1	Parameter list address (IQEPARAM)
2-12	Do not contain any information for use by the routine.
13	Problem program register save area address, if any
14	Return address
15	Entry point address
- When the asynchronous exit completes:
  - If you did not specify the STAB=DYN option on the CIRB macro, you can reuse the IRB when the IRB has completed execution. To know the IRB has completed, you can:
    - Make sure that the IRB has received control.
    - Obtain the local lock and check the RBFACTV bit.

If the IRB has received control and the RBFACTV bit is off, then the IRB has completed execution, and you can reuse the IRB.

  - If the caller specified the SVAREA and STAB=DYN options on the CIRB macro, the problem program register save area is freed.
  - If the caller specified the STAB=DYN option of the CIRB macro, the IRB and IQE are freed. If the IRB ran under a task other than the task that issued CIRB, you need to have specified the BRANCH=YES option of the CIRB macro, having placed the address of the TCB of the task under which the IRB ran into register 4 before issuing CIRB, in order for the freeing to succeed.
- If your program provided the IQE, you can update the IQE if you hold the local lock and have verified that the IRB has received control.

Using the SCHEDXIT macro to schedule an IRB

You can use the SCHEDXIT macro to schedule an asynchronous exit to run. The calling program supplies an initialized interrupt queue element (IQE) to the SCHEDXIT macro. The IQE identifies the task the exit routine is to run under and the associated IRB. It also contains information about the exit routine's characteristics.

The caller must initialize the IQE to define to the system the task under which the exit routine is to run. The fields to be initialized are:

- IQEPARAM -- optional address of the parameter list to be passed to the exit routine.
- IQEIRB -- address of the IRB as returned in register 1 by the CIRB macro.
- IQETCB -- address of the TCB for the task under which the user's exit routine is to run.

When IRB/IQE initialization is complete, the caller should invoke SCHEDXIT to queue the request (IQE) to the appropriate system asynchronous exit queue. The caller can invoke SCHEDXIT in one of two ways, depending primarily on the location of the IQE:

- If the IQE is in 31-bit storage, the caller must invoke the SCHEDXIT macro. Detailed information about using the SCHEDXIT macro is in *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.
- If the IQE is in 24-bit storage, the caller can use the SCHEDXIT macro only if the IQE address passed is a clean 31-bit address (that is, the high-order byte of the address is zero). Otherwise, the caller must use branch entry, where the branch entry point address is in field CVT0EF00 of the communications vector table (CVT).

  For branch entry, the interface is defined as follows:

Register	**Contents**
0	Irrelevant
1	Twos-complement IQE address
2-13	Irrelevant
14	Return address
15	Irrelevant

  Upon return, the registers contain the following:

Register	Contents
0	Used as a work register by the system
1	True (non-complemented) IQE address
2-13	Unchanged
14, 15	Used as work registers by the system

Whether the caller uses the SCHEDXIT macro or branch entry, the caller must:

- Hold the local lock
- Have addressability to the address space in which the exit routine is to be dispatched
- Be in supervisor state with PSW key zero.

System processing to run an asynchronous exit

Once scheduled by the SCHEDIRB or SCHEDXIT macro, the asynchronous exit routine is logically ready to be dispatched.

If the target TCB was waiting, the system brings the target TCB out of the wait state, and the exit routine processes when the task gets dispatched next (according to its dispatching priority). If the target TCB was running, the exit routine processes after the task gets preempted for reasons unrelated to the exit, such as a page fault, a WAIT macro, or expiration of the task time to run on the processor.

If you used SCHEDIRB with the RBPTR=*rb addr* option to use directed IRBs, the system allows the directed IRB to run directly prior to the specified RB. The system processes the asynchronous exit routine in the next dispatch of the task, unless a subsequent non-directed IRB was scheduled for the same task. In that case, the directed IRB's exit routine will still get control just before its target RB, but the target RB might not run next. A subsequent IRB could get control first.

Linkage stack considerations for asynchronous exit routines

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

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

Establishing a timer disabled interrupt exit

The system provides a function called set DIE that allows a user-written program to establish a disabled interrupt exit (DIE) routine. The DIE routine gains control asynchronously after a specified real time interval has elapsed.

The set DIE function is available only to programs executing in supervisor state with PSW key zero. The set DIE function allows users to initiate a real time interval by branching to the set DIE system service. When the time interval expires, the user's DIE routine gains control as an extension of the timer second level interrupt handler. It is also possible for a user to set a new time interval from the DIE routine.

Although a program can have an unlimited number of outstanding time intervals at one time, storage and system performance considerations may impose practical and reasonable limits.

Note: The time during which a DIE routine is executing is not charged to the job step time of the interrupted address space.

The caller of the set DIE service routine can be executing in either task control block (TCB) or service request block (SRB) mode, but must be in PSW key zero and supervisor state. The entry point to the set DIE service routine is in field TPCSDIE in the timer supervision work area mapped by macro IEAVVTPC. The address of this work area is in CVT field CVTTPC.

DIE routines execute in an MVS-recognized disabled state, and therefore do not need to obtain the CPU lock. They must, however, comply with the special characteristics and restrictions that apply only to DIE routines, as described in this information.

Note that except where explicitly documented, register descriptions apply to 32-bit GPRs. The exception is for AMODE 64 cases, either invoking a service in AMODE 64 or where an exit routine gets control in AMODE 64.

The caller of the set DIE service routine must provide the following input environment.

1. Register 1 must contain the address of a user-supplied timer queue element (TQE) whose fields are available from the IHATQE mapping macro. This user TQE must:
   - Be a contiguous block of 128 bytes aligned on a double word boundary.
   - Reside in SQA.
   - Include the following field initialization:

     TQEAID -- zero or a valid ASID, important in case of an address space failure (see “Obtaining and freeing the TQE” on page 470).

     TQEVAL -- the desired real time interval (a 64 bit unsigned binary number with bit 51=1 microsecond).

     TQEAMODE bit -- set to 1 if you wish +o indicate that the address of the user's DIE in TQEEXIT is pointer defined.

     TQEEXIT -- address of the user's DIE. If the TQEAMODE bit is set to 1, the high-order bit of this field, TQEXMODE, must indicate the addressing mode of the user's DIE. If the user's DIE is to execute in 24-bit addressing mode, TQEXMODE=0; if the user's DIE is to execute in 31-bit addressing mode, TQEXMODE=1.
   - Have all the other fields cleared to zero.
2. Registers 2 - 12 must be parameter registers whose input values will be restored in the same registers on entry to the DIE routine.
3. Register 14 must contain the caller's return address.

Loss of the contents of register 1 and 11-13 occurs upon return from the set DIE service routine. Register 15 contains a return code as follows:

Code	Meaning
0	The TQE was successfully enqueued onto the system's real time queue.
4	Failure - needed clocks are unavailable.

The set DIE service routine does not establish its own recovery routine. Any program calling the set DIE service routine should have its own FRR or ESTAE routine. A program check occurs in the set DIE service routine if the caller is not both in PSW key zero and in supervisor state.

The DIE routine gains control under the system's FRR on the current stack. The DIE itself can optionally establish its own FRR, which should terminate by percolation to let the timer supervision FRR gain control. For additional information on the environmental factors on entry to an FRR, see "Environment on entry to an FRR" on page 381.

DIE characteristics

The entry environment is:

Environmental factor	**Requirement**
Authorization:	Supervisor state and PSW key 0.
Dispatchable unit mode:	Task or SRB
Cross memory mode:	Any PASN, any HASN, any SASN
AMODE:	24-bit or 31-bit depending on TQEXMODE bit when the DIE was set.
ASC mode:	Primary
Interrupt status:	Disabled for I/O and external interrupts
Locks:	The local, CMS, and/or CPU locks may be held.
Control parameters:	No requirement.

Register contents upon entry are as follows:

- The contents of register 0 is unpredictable.
- Register 1 contains the address of the TQE. At this time the TQE is not enqueued upon the real time queue. Fields TQETCB and TQEASCB respectively contain a TCB address and an ASCB address, if previously set by the user on entry to the set DIE service routine.
- The low-order words of registers 2 - 12 are as they were upon entry to the set DIE service routine (or as changed by a previous DIE entry -- see "DIE execution" on page 469).
- The contents of register 13 is unpredictable.
- Register 14 contains the return address.
- Register 15 contains the entry point of the DIE routine.
- The contents of floating point registers are unpredictable.
- The contents of vector registers (VRs) are unpredictable. The DIE must not use VRs.
- Access registers and the high halves of general purpose registers are unpredictable on entry to a DIE routine.

While a system program has a TQE enqueued upon the real time queue, it must ensure that the associated DIE routine is available for the timer SLIH (second level interrupt handler) to access from any address space. Additionally, because the DIE is entered disabled, its code must be resident or fixed to avoid a page fault at entry.

Exit from the DIE routine

Must be to the address specified in register 14. This exit must also occur in supervisor state with PSW key zero, and disabled. The routine needs not save or restore any part of the general purpose or access registers that it uses.

On entry to the DIE, the contents of the Floating Point Registers (FPRs) and Floating Point Control (FPC) register are unpredictable. If the DIE uses the FPRs or FPC register, it must ensure that these registers are restored to the values they had on entry. See “Floating point protocol for user exits” on page 477.

On entry to the DIE, the contents of the Vector Registers (VRs) are unpredictable. The DIE is not allowed to use VRs.

DIE execution

Must be like the execution of an interrupt handler. Specifically, the DIE routine executes under the following restrictions:

- The DIE must be capable of executing in any address space because the timer interruption may occur while any address space enabled for external interruptions is executing.
- The DIE may not reference any private storage areas.
- The DIE may not use ALET=2 in AR ASC mode.
- The DIE must execute disabled. Hence, it may not cause a page fault.
- The DIE may not request a local lock or the CMS lock because these are suspend locks and might therefore already be in use. Furthermore, the DIE routine may not assume whether or not these locks are held upon entry.
- The DIE may not execute any SVCs.

The DIE routine may re-enqueue the TQE to set another real time interval by using the timer's TQE ENQUEUE routine (whose entry point is in CVT field CVTQTE00). The DIE routine must issue SETLOCK OBTAIN,TYPE=DISP,MODE=UNCOND upon entry to the TQE ENQUEUE routine.

The input environment for the TQE ENQUEUE routine must be as follows:

- Supervisor state, key zero.
- Register 1 must contain the address of the TQE supplied to the DIE routine. Only the following TQE fields can be changed.

  TQEVAL -- This field should contain the clock comparator value for the next interruption. This value is equivalent to the desired interval added to the value in TQEVAL when the DIE routine was entered. Alternatively, TQEVAL can be calculated by adding the desired interval to the current TOD clock reading (as obtained by a STCK instruction). The choice of which method to use is further discussed under “Clock Failure”.

  TQEAMODE bit -- set to 1 to indicate that the address of the user's DIE in TQEEXIT is pointer defined.

  TQEEXIT -- This field should contain the new address if a DIE routine address different from the current one is desired. Otherwise the field should remain unchanged. If the TQEAMODE bit is set to 1, the high-order bit of this field, TQEXMODE, must indicate the addressing mode of the user's DIE. If the user's DIE is to execute in 24-bit addressing mode, TQEXMODE=0; if the user's DIE is to execute in 31-bit addressing mode, TQEXMODE=1.

  TQEDREGS -- If the parameter values in the low halves of registers 2 - 12 are to be changed for the subsequent DIE routine entry, the new values should be set in this 11-word field.
- Register 2 must contain the caller's return address.

Upon return from the TQE ENQUEUE routine, all registers are as they were on entry except for registers 13 and 15. The DIE routine must issue SETLOCK RELEASE,TYPE=DISP.

Although the set DIE function is similar to the TQE ENQUEUE function, the routines differ in the following respects:

- Although TQE ENQUEUE expects an already established and fully initialized TQE as input, the set DIE service routine completes the user-supplied TQE to make it acceptable to timer supervision.
- For TQE ENQUEUE, TQEVAL in the TQE must be set to the clock comparator value for the next interruption. With the set DIE service routine, it must be set to the desired interval. The set DIE service routine then converts it to the proper clock comparator value.
- TQE ENQUEUE assumes that the clocks are functioning correctly. The set DIE service routine must use the clocks directly and therefore verifies (rather than assumes) that the clocks are functioning correctly. The set DIE service routine is therefore capable of advantageously using alternate clocks in a multiprocessing environment in which one or more clocks have failed.

Timer queue element control

The major aspects of controlling the timer queue element (TQE) associated with the user's DIE routine are:

- Obtaining and freeing the TQE
- Serializing the use of each TQE
- Time-of-day clock failure
- Interval cancellation

Descriptions of each of these aspects follow.

Obtaining and freeing the TQE

Is your responsibility as user of the set DIE function because the TQE resides in SQA. Thus, you must explicitly free the TQE when it is no longer necessary and (with one exception) in error situations as well. Timer supervision frees a TQE for you for a failing address space only if the TQE is enqueued on the real time queue and has field TQEAID set to the ASID of the failing address space.

Before freeing the TQE, however, you must ensure that it is not currently on the real time queue. There are several ways to accomplish this:

- Always free the TQE in the DIE routine because it is never on the real time queue when the routine receives control.
- Before freeing the TQE, use timer supervision's TQE DEQUEUE routine. This routine either removes the TQE from the real time queue or, if the TQE is not on the queue, takes no action.

Note:

1. You must not alter the TQE (other than in the fields previously described).
2. The interface for the TQE DEQUEUE routine is described in “Interval Cancellation”.

Serializing the use of each TQE

Is also your responsibility. Serialization includes the execution of the set DIE service routine, TQE ENQUEUE, and TQE DEQUEUE routines for a given TQE because these routines update the supplied TQE. Never update a TQE, however, while it is on the real time queue.

Clock failure

Can keep a DIE routine from receiving control. If a clock required by a DIE routine's TQE fails while the TQE is on the real time queue, timer supervision leaves the TQE on the queue, thereby denying control to the DIE routine. To permit the DIE routine to receive control, a properly functioning TOD clock and clock comparator must be varied online. For this remedy to work, the DIE routine must be in resident or fixed storage as long as its TQE is on the real time queue. These storage locations make the DIE routine available to the system from any address space.

When the DIE routine gains control under these circumstances, the clock comparator value in TQEVAL could be behind the TOD clock. If the DIE routine re-enqueues the TQE on each successive entry and adds a new interval to TQEVAL, then the DIE routine gains control each time, immediately upon enablement of the external interruptions. This sequence continues until the value in TQEVAL is equal to the TOD clock value. To avoid this synchronization loop, the DIE routine can calculate the new TQEVAL as the sum of the new interval plus the current TOD clock value. This method, however, requires that the DIE routine contain error recovery code in case the STCK instruction fails due to a bad TOD clock in the executing processor.

Interval cancellation

Can occur by using timer supervision's TQE DEQUEUE routine. This routine removes a specific TQE from the real time queue and resets clocks if necessary. The entry point to the TQE DEQUEUE routine is in CVT field CVTQTD00. Entry to this routine must be by branch entry, in supervisor state, with PSW key zero. You must be in primary ASC mode and the primary address space must be the home address space (PASH=HASH). Before entering the TQE DEQUEUE routine, you must issue SETLOCK OBTAIN,TYPE=DISP,MODE=UNCOND. The input environment is as follows:

- Register 1 must contain the address of the TQE to be dequeued.
- Register 2 must contain the caller's return address.

Upon return, all registers except 13 and 15 are the same as they were on entry. Register 15 contains a return code:

Code	Meaning
0	The TQE was on the queue and has been removed.
4	The TQE was not on the queue.

Using dynamic exits services

The CSVDYNEX macro provides exits and controls their use; it also provides a way for you to associate one or more exit routines with those exits. You might be familiar with system installation exits that offer your programs an opportunity to interrupt the system's processing for any number of good reasons; generally the reason is for the system to obtain information on which to base its processing. Often, however, programs or vendor products take advantage of an installation exit to do processing of their own. CSVDYNEX allows you to define exits and control their use just as the system does when it offers installation exits.

The dynamic exits facility is a set of services implemented by:

- The EXIT statement of the PROGxx parmlib member. The EXIT statement allows an installation to add exit routines to an exit, delete an exit routine for an exit, change the state of an exit routine, change the attributes of an exit, and undefine an implicitly defined exit.

  The PROGxx EXIT statement interacts with the PROG=xx parameter of IEASYSxx and the SET PROG=xx command. At IPL, operators can use PROG=xx to specify the particular PROGxx parmlib member the system is to use. During normal processing, operators can use the SET PROG=xx command to set a current PROGxx parmlib member. See *z/OS MVS Initialization and Tuning Reference* for information about the PROGxx parmlib member.
- The SETPROG EXIT operator command. This command performs the same functions as the EXIT statement of the PROGxx parmlib member. See *z/OS MVS System Commands* for information about the SETPROG EXIT command.
- The CSVDYNEX macro.

An installation can use any of these methods to control dynamic exits. For example, an exit routine can be associated with an exit using the CSVDYNEX ADD request, the SETPROG EXIT,ADD operator command, or the EXIT statement of PROGxx.

Through the CSVDYNEX macro you can define an exit, control its use, and associate exit routines with it. None of these actions require a system IPL. The macro is of interest to two kinds of applications:

- The application that wants to offer an opportunity for another program, an exit routine, to intercept the application's processing. This application, known as the exit provider, defines the exit and calls the exit routine or routines.
- The application that wants one of its own routines to get control through an exit defined through CSVDYNEX. This application is known as the **exit associator**.

  CSVDYNEX, for example, allows you to associate one or more routines you are currently using, or plan to develop, with the existing SMF and allocation installation exits. Those exits have been defined to the dynamic exits facility.

The same application might be both exit provider and exit associator.

CSVDYNEX terminology

An **exit** is a set of information defined by the exit provider. That set of information includes:

- Characteristics of (or rules for) exit routines that are to get control at the exit point
- Directions for how the system is to transfer control to an exit routine, process the exit routine, and handle recovery.

Control passes to exit routines when the exit provider issues a call; the location of that call request is known as the **exit point**. The exit provider code defines an exit and later issues the call, as the following diagram shows:

```
exit provider setup

   _____
   _____

   define the exit

   _____

exit provider processing

   _____
   _____            exit point
   _____          /                          exit routine
   issue the call v--------------------------   ______
   _____                                         ______
   _____                                         ______
                                                  return
```

You will find the macro easier to use if you understand some facts about exits, exit routines, and callers in CSVDYNEX terms.

An exit that is defined through CSVDYNEX:

- Has a name unique within the system
- Can have one or more exit routines associated with it
- Is invoked by a caller
- When invoked, causes exit routines associated with it to run
- Exists over a defined span of time: for the life of the defining task, the life of the defining address space, or the life of the IPL
- Consists of **exit criteria**, the characteristics of (or rules for) exit routines that can get control when the exit is invoked
- Can be enabled for **FASTPATH processing**, which means there is less system processing
- Can have its definition removed through CSVDYNEX.

An exit routine that is associated through CSVDYNEX with an exit:

- Can be in an active or inactive state. An active routine is called; an inactive routine is not.
- Can have any valid exit routine name
- Can be added to an exit (or associated with an exit) at any time; likewise, it can be deleted from the exit
- Will not be called if it does not conform to the exit criteria specified when it was added to the exit
- Can get control with specific GPRs containing specific information
- Can return information to the caller in registers and a caller-provided parameter list according to the documented interface.

A caller (that is, the exit provider) that invokes an exit through CSVDYNEX:

- Causes the active exit routine or routines associated with the exit to run
- Processes the return codes from the exit routines
- Receives information from the exit routines in a **return area**
- Receives information about exit routine processing
- Can specify what the contents of certain GPRs are to be when the exit routines get control
- Can request that the CALL request for the exit have FASTPATH processing.

You can do three main tasks through the CSVDYNEX requests: defining an exit, calling an exit routine, and adding an exit routine to an exit. The first parameter on the CSVDYNEX macro specifies the request. For example, the ADD request is CSVDYNEX REQUEST=ADD, with appropriate parameters. The requests, listed by task, are as follows:

- Defining an Exit; see “Defining an exit” on page 474
  - DEFINE request
  - UNDEFINE request

- ATTRIB request
- Calling an Exit Routine or Routines; see “Calling an exit routine or routines” on page 476
  - CALL request
  - RECOVER request
  - QUERY request with QTYPE=CALL
- Associating an Exit Routine with an Exit; see “Associating an exit routine with an exit” on page 480
  - ADD request
  - MODIFY request
  - DELETE request
  - QUERY request with QTYPE=ADD

An additional request, the LIST request, is useful for obtaining information about the exits that are defined through the CSVDYNEX macro.

Vector Register protocol for user exits

Your exit routine will receive control with the caller's Vector Registers (VRs). If your exit routine uses any of the VRs 8-23, it must be sure to save and restore them.

Defining an exit

Defining an exit means telling the system the set of information that comprises the exit. You define the exit through the CSVDYNEX DEFINE request. A scenario of an exit provider defining an exit named EX1 would look like this:

```
exit provider setup
    _______

    CSVDYNEX REQUEST=DEFINE,EXITNAME==CL16'EX1'...
    _____
exit provider processing
    _____
    _____
    the call ------------------------  routine or routines
                                       associated with the
                                       exit named EX1
```

Questions you need to ask when you define the exit are:

- **What is the name of the exit?**

  The 16-character name must be unique within the system. Each CSVDYNEX request requires that you specify the name of the exit.
- **What should the addressing mode of the exit routines be?**

  If you are providing data above 16 megabytes, you might require that exit routines be AMODE 31.
- **Should the exit routines be reentrant?**

  If an exit routine can get control from more than one task or SRB before returning to the caller, require that exit routines be reentrant.
- **How long is the exit to remain in effect?**

  The exit provider is likely one task of many tasks that constitute the application or subsystem. The exit provider must decide whether the exit is to remain for the life of the exit provider's task, for the life of the exit provider's address space, or if it is to persist throughout the IPL. The PERSIST parameter choices are TASK, ADDRESSSPACE, or IPL, as follows:

- If the exit provider's task ends and PERSIST=TASK, then the system deletes the exit.
- If the address space containing that task ends and PERSIST=ADDRESSSPACE (or PERSIST=TASK), then the system deletes the exit.
- If the address space containing that task ends and PERSIST=IPL, then the exit remains (in which case frequently the application would be restarted).

- **How many times should the system continue to call an exit routine that consistently ends abnormally?**

  On the ABENDNUM parameter, you tell the system how many times an exit routine can abnormally end (and not retry from its recovery routine) before the system changes the state of the exit routine to inactive (that is, before the system stops calling it). On the ABENDCONSEC parameter, you describe how the system is to do the counting: does it count consecutive abnormal endings, or does it count by simply tallying the number of abnormal endings, ignoring the times the exit routine runs successfully? For example, suppose ABENDNUM has the value 2; if you then code:

  - ABENDCONSEC=YES, the system stops calling the exit routine after a total of 2 consecutive abnormal endings; in other words, the system resets the count to zero if the exit routine runs successfully.
  - ABENDCONSEC=NO, the system stops calling the exit routine after a total of 2 abnormal endings have occurred. An abend is counted when both of the following conditions exist:
    - The exit routine does not provide recovery, or the exit routine does provide recovery but percolates the error
    - The system allows a retry; that is, the recovery routine is entered with bit SDWACLUP off.

- **How should the system process the return codes from the multiple exit routines associated with the exit?**

  Each exit routine that runs at an exit has return information, including return and reason codes. On the DEFINE and CALL requests, you tell the system which routine's return information is to be placed into an area called the return area. This topic, which requires your understanding of parameters on the DEFINE and CALL requests, is described in “Returning information from multiple exit routines” on page 477.

- **Is the exit to be enabled for FASTPATH processing?**

  The topic of FASTPATH Processing is covered in “FASTPATH processing.”

FASTPATH processing

FASTPATH processing means that the system does not provide as much function, and thus the overall processing time is less. In particular:

- Exit processing runs in the caller's PSW key
- The exit routines run in the caller's state: supervisor or problem
- The system does not provide recovery
- The caller must provide a work area for the system to use if exit routines abnormally end
- The system does not use a security program, such as RACF, to do authorization-checking.

FASTPATH processing occurs when the DEFINE request enables the exit for FASTPATH processing and the CALL request specifies that exit routines are to have FASTPATH processing. Recovery for routines with FASTPATH processing is described in "Recovery for the CALL request" on page 479.

IBM recommends that you do not use FASTPATH processing except when performance is a key consideration.

Removing the definition of an exit

The definition of the exit is removed through the UNDEFINE request. When a definition is removed, exit routines associated with that exit remain associated. In this case, the undefined exit is sometimes called an **implicitly-defined exit**.

Calling an exit routine or routines

The CALL request is done by the exit provider; it can also be done by other applications. At a CALL request, the system passes control to the active exit routine or routines that are associated with the exit. If multiple exit routines are associated with the exit, control passes to one routine after another, in no predictable order. If you require that the exit routines run in a particular order, you must do the routing. The CALL request does not include the name of an exit routine or routines; the system knows which routines are associated with the exit. The scenario of calling the exits that are associated with the exit EX1 would look like this:

```
exit provider setup

   _____
   _____
   _____

   CSVDYNEX REQUEST=DEFINE,EXITNAME==CL16'EX1'...
   _____

exit provider processing

   _____

   CSVDYNEX REQUEST=CALL,EXITNAME==CL16'EX1'...  routine or routine
   _____                                    associated with the
                                            exit named EX1
```

Specifying the CALL request is somewhat determined by the definition of the exit. For example, you cannot issue a call specifying that the routines get FASTPATH processing if the exit is not enabled for FASTPATH processing. In other words, you cannot specify FASTPATH=YES on the CALL request if the exit was defined with FASTPATH=NO. However, if the exit was defined with FASTPATH=YES, you can specify either FASTPATH=YES or FASTPATH=NO on the CALL request. There is no requirement that all CALL requests be of the same FASTPATH type.

Questions you need to ask when you call an exit are:

- **How is the system to return information to the caller?**

  One of the decisions you make on the CALL request is what return information the system returns. Does it return the return information from all exit routines or from one exit routine. Parameters on the CALL request are RETINFO, RETAREA, and RETLEN.

  – RETINFO specifies whether the caller receives return information from the exit routine with the LOWEST return code (RETINFO=LOWEST), from the exit routine with the HIGHEST return code (RETINFO=HIGHEST), from all exit routines (RETINFO=ALL), or from the last exit routine to run (RETINFO=LAST).

  – RETAREA and RETLEN tell the system where the return area is and how large it is.

If you specify LOWEST or HIGHEST on RETINFO, you can also specify related parameters on the DEFINE request. For more information about this topic, see "Returning information from multiple exit routines."

- **Is the CALL request for the exit to have FASTPATH processing?**

  To have FASTPATH processing, the definer of the exit must have enabled the exit for FASTPATH processing. Then, the caller can request that the CALL request for the exit have FASTPATH processing. For more information, see "FASTPATH processing" on page 475 and "Recovery for the CALL request" on page 479.

- **Are the GPRs to contain certain values at entry to the exit routines?**

  You most likely want the exit routines to get control with certain values placed in certain GPRs. The RUB parameter provides an area where you tell the system which GPRs are to contain the data and what the data is to be. *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* describes the format of the RUB and contains an example of its use.

- **Did any exit routines get control at an exit?**

  Appropriate return and reason codes indicate that an exit routine ran.

- **Did all exit routines get control at an exit?**

  If one or more one exit routines didn't run (usually because the return area was not large enough to contain all the return information):

  - The appropriate reason code indicates that fact.
  - The area provided on the NEXTTOKEN parameter contains a token that identifies the next exit routine that the system is to call.
  - The return area contains return information for any exit routine that ended, or contains the return information that the DEFINE and ADD requests specified. You provided this area for the system to return information needed if not all exit routines were called.

  The caller can reissue the CALL request, using the value in NEXTTOKEN. The system will pass control to the exit routine that is the next to run.

Floating point protocol for user exits

Your exit routine will receive control with the caller's Floating Point Register and Floating Point Control register. If your exit routine uses any of the Floating Point Registers or the Floating Point Control register it must be sure to save and restore them.

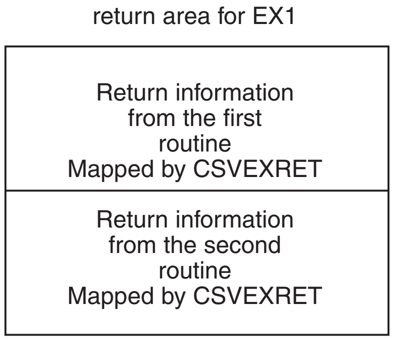
Ensuring that exit routines exist at the CALL

If the work involved in setting up information for the CALL request is costly, you might issue the QUERY request with the QTYPE=CALL parameter to find out whether any exit routines are associated with the exit. This step would help you avoid the steps that lead up to a CALL request. Here is an example, in pseudocode, of using the QUERY request:

```
Issue REQUEST=QUERY with TYPE=CALL
IF retcode indicates "there are exit routines" THEN
  set up parameter list
  Issue REQUEST=CALL
ELSE
  do "no exit routine" default processing
ENDIF
```

Returning information from multiple exit routines

On the CALL and RECOVER requests, CSVDYNEX provides a return area for the caller to receive information relating to successful and unsuccessful processing of one or more exit routines. As it processes an exit, the system places information (called **return information**) in that return area. Before you issue the CALL request, you obtain the storage for the return area; on both the CALL and RECOVER requests, you specify its address on the RETAREA parameter and its length on the RETLEN parameter. The following illustration shows the return information from two exit routines associated with EX1, where the CALL specified RETINFO=ALL.


return area for EX1
Return information from the first routine Mapped by CSVEXRET
Return information from the second routine Mapped by CSVEXRET

For description of CSVEXRET, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

The caller might not want to receive all the return information from all the exit routines. The RETINFO parameter on the CALL request tells the system which information the caller should receive. If the caller wants the system to receive return information from only one of the potentially many exit routines, the DEFINE request with RCFROM, RCTO, RCCOMPARE, and RCCVAL parameters can specify how the system is to choose which return information to return. The process of choosing involves **merging** return codes.

On the CALL request, you choose which return information the system places in the return area:

- The caller receives return information from all routines (the RETINFO=ALL parameter). In this case the system does not need to merge the return codes and none of the RCxx parameters on the DEFINE request apply. The caller must provide a large enough return area for all the information; if the area becomes full, the system does not call any more exit routines and returns to the caller with a token in the NEXTTOKEN parameter. The caller can reissue the CALL request, using the NEXTTOKEN parameter. The system then continues calling the exit routines where it left off.
- The caller receives return information from only the last routine (the RETINFO=LAST parameter). In this case the system does not need to merge the return codes and none of the RCxx parameters on the DEFINE request apply.
- The caller receives return information from the exit routine that returns with the highest return code or the exit routine that returns with the lowest return code. In this case, the RCFROM, RCCOMPARE, RCTO, and RCCVAL parameters on the DEFINE request tell the system to compare and possibly modify return codes, before choosing the lowest or highest. “Merging return codes” contains more information about this topic.

Merging return codes: This information describes the situation in which the CALL request uses the RETINFO=HIGHEST or RETINFO=LOWEST parameters. In this case, the DEFINE request must tell the system how to merge the return codes before it chooses which return information to return to the caller. The relevant parameters on the DEFINE request are as follows:

- RCFROM specifies the RCFROM return code. The system compares the actual return code from each exit routine to the RCFROM return code, using the comparison designated by RCCOMPARE.
- RCCOMPARE indicates the type of comparison the system is to make. For example, if a definer used RCCOMPARE=LT, the system would compare the actual return code to the RCFROM return code. If the actual return code is less than the RCFROM return code, the system replaces the actual return code with the value specified on RCTO.

The best way to describe the use of these parameters is through an example. Suppose the caller wants to handle only the return code with a value of 4. You would code the following values:

- On the CALL request, you would specify RETINFO=HIGHEST
- On the DEFINE request, you would specify:
  - The value 4 on RCFROM
  - The value 0 on RCTO
  - RCCOMPARE=NE

These settings tell the system to compare the return code from the exit routine with the number 4. If the return code is not 4, the system changes that value to 0. When all exit routines have completed, the system looks at all return codes (some with changed values) and returns the return information for the exit routine that had the highest return code.

If three exit routines get control at that exit, and their return codes are 0, 4, and 8, the system examines each code:

- 0 is not equal to 4, so it remains 0
- 4 is equal to 4, so it remains 4
- 8 is not equal to 4 so it is replaced with 0.

In this case, the highest return code is 4; the system places return information for that exit routine in the return area.

If two exit routines return with the same value in GPR15, the system places in the return area the return information from the first exit routine that had the value.

You can also use the ATTRIB request to change settings for how the system merges return codes.

Recovery for the CALL request

As definer of the exit and caller of exit routines, you need to know what to expect in the event that an exit routine abnormally ends. The CSVDYNEX service responds to such a failure according to whether or not the exit routines have FASTPATH processing:

- If FASTPATH processing is in effect (that is, the CALL request has FASTPATH=YES specified), the CSVDYNEX service does not provide recovery for a failing exit routine. The issuer of the CALL request must provide it. If an exit routine abnormally ends, control passes to the caller's recovery routine. That routine must issue the RECOVER request to complete the processing of the exit and return control to the system.
- If FASTPATH processing is not in effect and an exit routine abnormally ends, the CSVDYNEX service performs recovery and continues processing the exit,

perhaps passing control to the next exit routine. The CSVDYNEX service places information in the return area indicating that the routine abnormally ended.

When you define an exit, you can limit the number of abnormal endings an exit routine can have before the system stops calling it.

Recovery for exit routines with FASTPATH processing: FASTPATH processing requires the caller to write a recovery routine to cover abnormal endings for the exit routines. The recovery routine must contain the RECOVER request. On that request, the caller provides:

- The address of a work area (on the WORKAREA parameter)

  The CSVDYNEX service needs this area for its own use. The work area must be the same area specified on the CALL request.
- The address of the SDWA (on the SDWA parameter)

  The caller must specify the address of the SDWA associated with the abnormally ended exit routine on the SDWA parameter. The caller can find this address in GPR1 when it gets control from RTM in its recovery routine. If the SDWA was not passed (GPR0 has a value of 12), the caller must provide an SDWA address of zero.
- An area for the NEXTTOKEN (on the NEXTTOKEN parameter)

  In this area, the CSVDYNEX service returns a token that identifies the exit routine that abnormally ended.
- A return area (on RETAREA and RETLEN parameters)

  In this area, the CSVDYNEX service stores the return information of the abnormally ended exit routine.

Associating an exit routine with an exit

You associate an exit routine with an exit using the CSVDYNEX ADD request. If the exit routine is not in the LPA list or in the nucleus, and you did not use the MODADDR keyword, the system loads the exit routine into common storage.

It is important that the exit associator know the exit criteria established by the exit definer. To find out the exit criteria, check the requirements imposed by the exit definer (the system, for SMF and allocation exits; or a vendor or customer application).

When you use the ADD request, you tell the system the name of the exit with which you are associating the exit routine. You also tell the system where to find the exit routine code. Other important questions you need to ask are:

- **Do you want the exit routine to be called only if a certain address space is primary (STOKEN parameter), or if a certain job (JOBNAME parameter) is running?**

  You can change the STOKEN or JOBNAME specification through the MODIFY request.
- **Do you want the exit routine to be called even if on previous calls it has abnormally ended?**

  The ADDABENDNUM and ABENDCONSEC parameters on the ADD request are similar to the ABENDNUM and ABENDCONSEC parameters on the DEFINE request. On the ADD request, you can override the settings that were defined on the DEFINE request.

The following scenario shows the associating of an exit routine named RTN1 with an exit named EX1.

```
exit associator code
 _________
 _________

CSVDYNEX REQUEST=ADD,MODNAME==CL8'RTN1',
           EXITNAME==CL16'EX1'...
 _________
```

You can associate an exit routine with an exit that does not exist (perhaps because it has not been defined yet). In this case, the system maintains the association; however, the exit routine does not get control until the exit is defined and the CALL is issued.

You can change two of the parameter specifications of the ADD request. Through the MODIFY request, you can:

- Change the state of an exit routine from active to inactive, or vice versa
- Change the requirement that the exit routine runs while a specific job runs or while a certain address space is the primary address space.

Example of associating an exit routine with an installation exit: Although you might use the system command SETPROG to associate an exit routine with an installation exit, you can also accomplish this action through a program. The following example shows how you would associate the routine named MYMOD with the SMF installation exit known as IEFUJI, defined through the SYS statement in SMFPRMxx parmlib member. The load module is in data set MY.DSN.

```
         CSVDYNEX REQUEST=ADD,EXITNAME=LEX,
               MODNAME=LMOD,STATE=ACTIVE,DSNAME=LDSN,
               RETCODE=LRETCODE,RSNCODE=LRSNCODE,MF=(E,DYNEXL)
*
* Place code to check return/reason codes here
:
:
:
* Data Declarations
LEX      DC    CL16'SYS.IEFUJI'
LMOD     DC    CL8'MYMOD'
LDSN     DC    CL44'MY.DSN'
         CSVEXRET              Return code information
DYNAREA  DSECT
LRETCODE DS    F
LRSNCODE DS    F
         CSVDYNEX MF=(L,DYNEXL)
```

Ensuring that exit routines exist at the time of the association

If the work involved in issuing the ADD request is costly, you might issue the QUERY request with the QTYPE=ADD parameter to find out whether any exit routines are associated with the exit. This step would help you avoid unnecessary processing. Here is an example, in pseudocode, of using the QUERY request:

```
QTYPE=ADD:

  Issue REQUEST=QUERY with QTYPE=ADD
  IF the return code indicates "there are no exit routines" THEN
    (perhaps no PROGxx parmlib member associated a exit routine
      with this exit)
    Add a "default" module to the exit.
  ENDIF
```

Deleting an exit routine from an exit

When the exit routine is no longer needed, the exit provider should use the DELETE request to remove the association between the exit and the exit routine.

At a DELETE request, the system generally frees the storage that contained the exit routine once it determines that the exit routine is not in use. The following rules describe the system's freeing of storage:

- **It does not free storage**:
  - When the exit routine originally resided in the LPA list or in the nucleus.
  - When the exit routine was associated with the exit (the ADD request) using the MODADDR parameter.
  - When the exit has FASTPATH processing and the DEFINE request includes FORCE=NO (the default), and the exit definer specified PSW key 8 to 15.
- **It frees the storage immediately, without checking to see if the exit routine is in use**:
  - When the DELETE request is for a FASTPATH exit, the exit definer specified PSW key 8 to 15, and the DELETE request specifies FORCE=YES. (In this case, the issuer of the DELETE request must determine whether the exit routine is in use.)
- **In all other cases, it frees the storage after it determines that the exit routine is not in use.**

As you decide whether you need to free the storage, consider the effects of address space termination. If the exit routine resides in the private area of an address space (this can only happen if you use the MODADDR keyword), make sure that you use either the JOBNAME or STOKEN parameter, and that the exit itself is not called after the address space terminates. If you don't limit the use of the exit routine, the system could end up trying to call the exit routine after its storage no longer exists due to the termination of the job or address space.

Chapter 23. User-written SVC routines

This information explains how to provide a supervisor state service to an application, including how to do the following:

- Write SVC routines
- Insert SVC routines into the control program
- Screen subsystem SVCs.

Writing SVC routines

You can introduce user-written SVC routines into the control program whenever you IPL the system. When you write an SVC routine, you must follow the same programming conventions used by SVC routines supplied with the system. Five types of SVC routines are supplied, and the programming conventions for each type are different.

SVC routines, including user-written ones, can either be part of the resident control program (the nucleus), or be part of the fixed or pageable link pack area. Types 1, 2, and 6 SVC routines become part of the resident control program, and types 3 and 4 go into the link pack area. Before IPLing the system, you must place your SVC routine in SYS1.NUCLEUS or SYS1.LPALIB. You must also create, before IPLing the system, an IEASVCxx member in SYS1.PARMLIB with SVCPARM statements that describe the characteristics of your SVC routine.

SVC routines receive control with PSW key zero and in supervisor state. They must be reenterable and, if you want to aid system facilities in recovering from machine malfunctions, they must also be refreshable. If you write two or more SVC routines that must serialize with each other, use the locking facilities or the ENQ and DEQ macros.

When you insert an SVC routine into the control program, you specify which locks the routine will require. When an SVC routine receives control, it is normally enabled and it can be holding one or more locks. However, if you specified that the routine requires a CPU lock, the routine is disabled when it receives control. The routine is also entered in a disabled state if it is a type 6 SVC routine.

If the SVC needs to update the PSW in the current or prior RB, it should use the IEARBUP service. IEARBUP allows part or all of the PSW in RBOPSW to be updated. Updates made directly to RBOPSW may be ignored.

Type 6 SVC routines

You must define your user-written SVC routine as being one of the five valid types, including type 6. The type 6 SVC routine performs functions similar to the type 1 SVC routine. However, the type 6 routine offers performance advantages over the type 1. The type 6 SVC routine cannot require the LOCAL lock, as noted later.

The type 6 SVC also provides a more efficient way to change from TCB mode to SRB mode processing. The type 1 SVC must schedule an SRB, which then goes through queuing and dequeuing operations before it is eventually dispatched. The type 6 SVC, however, normally results in immediate scheduling and dispatching of the SRB.

Because a type 6 SVC routine executes under the control of the SVC first level interrupt handler (FLIH), it must preserve disablement and it cannot obtain any locks. When a type 6 SVC routine exits, it always returns to the SVC FLIH. There are three exit options for a type 6 SVC:

- Return to the caller directly
- Return to the dispatcher
- Dispatch an SRB (service request block)

To exit from a type 6 SVC routine, either issue the T6EXIT macro or use the original contents of register 14 as a return address. The use of T6EXIT results in the register conditions described in the following paragraphs.

If a type 6 SVC uses the RETURN=CALLER exit option on the T6EXIT macro, or if it returns by branching on register 14, registers 0, 1, and 15 are returned to the caller. The rest of the caller's registers are unchanged. All of the caller's registers are unchanged if the type 6 SVC uses the RETURN=SRB or DISPATCH exit option on the T6EXIT macro.

If a type 6 SVC uses the RETURN=SRB exit option on the T6EXIT macro, register 1 must point to an SRB. The SRBASCB field must indicate the current address space.

The system neither acquires nor releases any locks for type 6 SVCs. Because a type 6 SVC executes in an MVS-recognized disabled state, it has exclusive use of the processor. Because the type 6 SVC routine runs in disabled state, the routine can only reference data in non-pageable storage. Type 6 SVC routines should be short enough to minimize any adverse effect on performance and they should provide for recovery by using the SETFRR macro.

Non-preemptable SVC routines

You can use the SVCUPDTE macro or the IEASVCxx parmlib member to define a user-written SVC routine as non-preemptable for I/O interruptions. If a non-preemptable SVC routine sustains an I/O interrupt, the SVC, rather than the highest priority ready work, gets control when I/O processing is complete. The non-preemptable SVC cannot issue other SVCs and remain non-preemptable. If a non-preemptable SVC issues a STAX DEFER=NO macro, the SVC routine remains non-preemptable until it exits. For a description of the STAX macro and its syntax, see *z/OS TSO/E Programming Services*.

Programming conventions for SVC routines

Table 66 summarizes the programming conventions for the five types of SVC routines. Details about many of the conventions are in the reference notes that follow the figure. The numbers in the far right column of the figure correspond to the reference notes.

Table 66. Programming Conventions for SVC Routines

Conventions	Type 1	Type 2	Type 3	Type 4	Type 6	Reference Code
Part of resident control program	Yes	Yes	No	No	Yes	

Table 66. Programming Conventions for SVC Routines (continued)

Conventions	**Type 1**	**Type 2**	**Type 3**	**Type 4**	**Type 6**	**Reference Code**
Size of routine	Any	Any	Any	Any	Any	
Reenterable routine	Yes	Yes	Yes	Yes	Yes	1
Refreshable routine	No	No	Yes	Yes	No	2
Locking requirements	Yes	No	No	No	No	3
Entry point: Must be on a halfword boundary and must be the first instruction to get control. Need not be the first byte of the module						
Number of routine: Numbers assigned to your SVC routine should be in descending order from 255 through 200						
Name of routine	IGCnnn	IGCnnn	IGC00nnn	IGC00nnn	IGCnnn	4
Register contents at entry time: Registers 3, 4, 5, 6, 7, and 14 contain communication pointers; registers 0, 1, 13, and 15 are parameter registers						
Supervisor request block(SVRB) used	No SVRB exists	Yes	Yes	Yes	No SVRB exists	6
May issue WAIT macro	No	Yes	Yes	Yes	No	7
May suspend their caller	Yes	No	No	No	Yes	8
May issue XCTL macro	No	Yes	Yes	Yes	No	9
May pass control to what other types of SVC routines	None	Any	Any	Any	None	10
Type of linkage with other SVC routines	Not Applicable	Issue supervisor call (SVC) instruction	Issue supervisor call (SVC) instruction	Issue supervisor call (SVC) instruction	Not Applicable	11
Exit from SVC routine	Branch using return register 14	Branch using return register 14	Branch using return register 14	Branch using return register 14	T6EXIT or BR 14	12
Method of abnormal termination	ABEND	ABEND	ABEND	ABEND	ABEND	
Recovery	FRR	ESTAE or FRR	ESTAE or FRR	ESTAE or FRR	FRR	13

Reference Code	**SVC Routine Types**	**Reference Notes**
1	all	If your SVC routine is to be reenterable, you cannot use macros whose expansions store information into an inline parameter list.

Reference Code	SVC Routine Types	Reference Notes
2	3,4	Types 3 and 4 in the pageable LPA must be refreshable. Types 3 and 4 in the fixed LPA must be reenterable, but not necessarily refreshable.
3	all	The following conventions on locking requirements apply: • Type 1 SVC routines always receive control with the LOCAL lock held and must not release the LOCAL lock. Additional locks may be requested prior to entry via the IEASVCxx member of SYS1.PARMLIB or the SVCUPDTE macro or may be requested dynamically within the SVC routine. • Types 2, 3, and 4 may also request locks via the IEASVCxx member of SYS1.PARMLIB or the SVCUPDTE macro or may obtain them dynamically. • Types 1 and 2 may request that any locks be held on entry. Types 3 and 4 may only request that the LOCAL or LOCAL and CMS be held. • If no locks are held or obtained, or only suspend locks (LOCAL and CMS) are held or obtained, the SVC routine executes in supervisor state key zero, enabled mode. • If the CPU lock is held or obtained, the SVC routine executes in supervisor state, key zero, disabled mode. No SVCs may be issued. • SVCs may not take disabled page faults. Therefore, if the CPU lock is held, the SVC routines must ensure that any referenced pages are fixed. For types 3 and 4, all pages containing code must be fixed. • An FRR may be defined for any SVC routine that holds or obtains locks to provide for abnormal termination (see reference code 13). • Type 6 may not request any locks.
4	all	You must use the following conventions when naming SVC routines: • Types 1, 2, and 6 can have either IGCnnn or non-IGCnnn names. – When using an IGCnnn name (where nnn is the decimal number of the SVC routine), you must specify this name in an ENTRY, CSECT, or START instruction. – When using a non-IGCnnn name, you must add the name to the nucleus region through the NMLDEF macro or NUCLSTxx parmlib member. • Types 3 and 4 must be named IGC00nnn; nnn is the signed decimal number of the SVC routine. For example, SVC 251 would be IGC0025A and SVC 245 would be IGC0024E. The following conventions regarding type 3 and 4 SVCs are not enforced by SVC processing, but have traditionally been used to distinguish between the two types: • A type 3 SVC identifies a function that is contained in a single load module. • A type 4 SVC identifies a function that loads additional modules. You can identify these loaded modules as IGC01nnn, IGC02nnn,..., and IGC0xnnn. (IGC01nnn is the first module that IGC00nnn loads, IGC02nnn is the second module that IGC00nnn loads, and IGC0xnnn is the last module that IGC00nnn loads.)

Reference Code	SVC Routine Types	Reference Notes
5	all	Before your SVC routine receives control, the contents of all registers are saved. In general, the location of the register save area is unknown to the routine that is called. When your SVC routine receives control, the status of the general purpose registers is as follows: * Registers 0 and 1 are unchanged from when the SVC instruction was issued. * Register 2 is used as a work register by the system. * Register 3 contains the starting address of the communication vector table (CVT). * Register 4 contains the address of the task control block (TCB) of the task that called the SVC routine. * Register 5 contains the address of the supervisor request block (SVRB), if a type 2, 3, or 4 SVC routine is in control. If a type 1 or 6 SVC routine is in control, register 5 contains the address of the last active request block. * Register 6 contains the entry point address. * Register 7 contains the address of the address space control block (ASCB). * Registers 8-12 are used as work registers by the system. * Register 13 is unchanged from when the SVC instruction was issued. * Register 14 contains the return address. * Register 15 is unchanged from when the SVC instruction was issued. The access registers (AR) are unchanged from when the SVC instruction was issued. You must use register 0, 1, and 15 if you want to pass information to the calling program. The contents of General Purpose registers 2 through 14 are restored when control is returned to the calling program, Access registers are not.
6	2,3,4	When a type 2, 3, or 4 SVC routine receives control, register 5 contains the address of the SVRB.
	This SVRB contains a 48-byte “extended save area,” RBEXSAVE, for use by the SVC routine.	
7	2,3,4	You can issue the WAIT macro if you hold no locks. You can issue WAIT macros that await either single or multiple-events. The event control block (ECB) for single-event waits on the ECB list and ECBs for multiple-event waits must be in virtual storage. Type 6 SVCs may not issue WAIT but may issue SUSPEND.
8	1,6	Both type 1 and 6 SVC routines can issue SUSPEND RB=CURRENT to suspend their callers.
9	2,3,4	When you issue an XCTL(X) macro in a routine under control of a type 2, 3, or 4 SVC, the new load module must be located in the fixed or pageable link pack area.

Reference Code	SVC Routine Types	Reference Notes
		The contents of registers 2 through 13 are unchanged when control is passed to the load module; register 15 contains the entry point of the called load module. An SVC routine will get control with the caller's Floating Point Registers and Floating Point Control register. An SVC routine will get control with the caller's Floating Point Registers and Floating Point Control register. The S/390 linkage convention applies. For more information on the S/390, see "Linkage Convention" in *z/OS MVS Programming: Assembler Services Guide*.
10	all	No SVC routines except ABEND may be called if locks are held. ABEND may be called at any time.
11	all	No locks may be held. If locks are held, branch entry to SVCs is acceptable, or the locks may be freed, the SVC issued, and the locks obtained again.
12	all	Branch using return register 14 should be used. SVC routines that exit via BR 14 or T6EXIT must return control in the same state in which they received control, such as, key zero, supervisor state. Otherwise, if locks are held, returning to the system will result in abnormal termination. Note: To ensure that control is returned to the dispatcher, the SVC routine can load register 14 with the address in the CVTEXP1 field of the CVT before issuing BR 14.
13	all	If an SVC routine is entered with a lock held or if an SVC routine obtains a lock, it should specify a functional recovery routine (FRR) for as long as the lock is held (see SETFRR macro). The FRR receives control if an error occurs, and ensures the validity of the data being serialized by the lock; the FRR either recovers or releases the lock and continues with termination. If no FRR is specified, the recovery termination manager releases the lock and terminates the task. No cleanup of the data is performed. (Note that the lock is released before any STAI/ESTAI/ESTAE (or STAE) recovery routine is entered. If no locks are acquired for or by an SVC routine, then an ESTAE may be used to define your recovery processing (see ESTAE and SETRP macros).

Inserting SVC routines into the control program

There are two kinds of SVC entries: standard and extended. Standard user SVC entries are in the range of 200 - 255. For information on how to use extended SVC routines, see “Extended SVC routines” on page 491.

Standard SVC routines

To supply user-written SVC routines to the system, you place descriptions of your user SVC routines in SYS1.PARMLIB, and you place the actual routines in SYS1.NUCLEUS or SYS1.LPALIB. When the system is IPLed, the system translates the SVC definitions that you placed on SYS1.PARMLIB into SVC table entries. See “Customizing the nucleus region” on page 324 for more information on placing routines in the nucleus.

When the system is IPLed, the system searches for member names that it uses to build the SVC table. These member names have the form, IEASVCxx, where xx is the field specified by the SVC= option in the IPL system parameters.

In the IEASVCxx members, you code SVCPARM statements. The SVCPARM statements describe the properties and attributes of individual SVC routines. Each SVCPARM statement that you code describes a single SVC routine; it generates one entry in the SVC table. Using the SVCPARM statement, you specify the SVC number, type, entry point name, lock requirements, authorization level, and whether or not the SVC is preemptable. See *z/OS MVS Initialization and Tuning Reference* for a description of the SVC= and SVCPARM statements.

The user SVC entries, which are represented by the SVC numbers 200-255, are the only ones you are allowed to define. You should not attempt to modify SVCs that are in the range of 0-199. Doing so will cause unpredictable results. When you define an SVC with an SVCPARM statement, you define its type as type 1, 2, 3, 4 or 6. The system provides no SVC routines in the range 200-255. Therefore, unless the user defines some SVC routines in this range, execution of an SVC 200 through 255 will cause an abend.

Example of adding user SVC routines to system libraries

Once you have modified the SVC table for your SVC routine, you must add your SVC routine to an appropriate library. Type 1, 2, and 6 SVCs must be added as CSECTs to an IEANUC0x load module in the SYS1.NUCLEUS data set, or added to the nucleus region through the NMLDEF macro or NUCLSTxx parmlib member. Type 3 and 4 SVCs must be added as load modules to the SYS1.LPALIB data set. When adding these SVC routines, SMP/E will require them to be associated with an FMID that is known to the system. The examples that follow are using FMID(IPOFMID). If you want to use IPOFMID you will first have to run job SMPUCL in IPO1.JCLLIB to add it as a valid FMID that the system recognizes. If you have run job SMPUCL already, you do not have to run it again.

The following example uses SMP/E to add SVC 255 as a type 1, 2, or 6 SVC to member IEANUC01 in SYS1.NUCLEUS:

```
//ADDSVC1 EXEC IPOSMPE
//SMPCNTL    DD  *
  SET BDY(GLOBAL) .
  RECEIVE SELECT(SVC255V) SYSMOD .
  SET BDY(MVSTZN) .
  APPLY   SELECT(SVC255V) .
  LIST    SYSMOD(SVC255V) USERMOD .
//SMPPTFIN DD  *
++ USERMOD (SVC255V) .
++ VER(Z038) FMID(IPOFMID) .
++ MOD(IGC255) DISTLIB(USERLIB) LMOD(IEANUC01) .
* * * object deck for IGC255 follows here * * *
/*
//USERLIB  DD  DSN=USER.USERLIB,DISP=SHR
```

Note: The LMOD keyword on the above ++MOD modification control statement informs SMP/E that the module IGC255 is to be included in load module IEANUC01, the nucleus.

The following example uses SMP/E to add SVC 255 as a type 3 or 4 SVC to SYS1.LPALIB:

```
//ADDSVC3 EXEC IPOSMPE
//SMPEIN    DD  *
  SET BDY(GLOBAL) .
  RECEIVE SELECT(SVC255V) SYSMOD .
  SET BDY(MVSTZN) .
  APPLY   SELECT(SVC255V) .
  LIST    SYSMOD(SVC255V) USERMOD .
//SMPPTFIN DD DATA,DLM=$$
```

```
++ USERMOD   (SVC255V) .
++ VER(Z038) FMID(IPOFMID) .
++ JCLIN                        /* note inline JCLIN */ .
//LINKSVC EXEC PGM=IEWL,PARM='NCAL,LET,RENT,LIST,XREF'
//SYSLMOD  DD  DSN=SYS1.LPALIB,DISP=SHR
//USERLIB  DD  DSN=USER.USERLIB,DISP=SHR
//SYSLIN   DD  *
  INCLUDE USERLIB(IGC0025E)
  NAME    IGC0025E(R)
//SYSUT1   DD  UNIT=SYSALLDA,SPACE=(CYL,(1,1))
//SYSPRINT DD  SYSOUT=A
++ MOD(IGC0025E) DISTLIB(USERLIB) .
* * * object deck for IGC0025e follows here * * *
/*
$$
```

Once you have added your SVC routine to the appropriate system library and updated the IEASVC*xx* parmlib member, you will have to re-IPL your system to have this take effect. If SYS1.LPALIB was the system library you updated, you need to specify CLPA or MLPA in the IEASYS*xx* parmlib member at the next IPL.

Modifying the SVC table at execution time (SVCUPDTE macro)

After the IPL, the SVC table can be dynamically modified by authorized users via the SVCUPDTE macro. For example, authorized subsystems such as VTAM® can alter the SVC table when the subsystem starts and restore the table when the subsystem terminates. For additional flexibility, the EPNAME and EXTRACT parameters of the SVCUPDTE macro allow the authorized user to dynamically associate SVC numbers with entry points of SVC routines.

NOT Programming Interface information

An SVC update recording table is maintained in parallel with the SVC table. This table provides a record of changes to the SVC table. Entries are created whenever a change is made to the SVC table with IEASVCxx parmlib member statements or the SVCUPDTE macro.

End of NOT Programming Interface information

Intercepting an SVC routine

When you execute an SVC instruction, the unique program to which control is passed is called the SVC routine. A common programming technique is to intercept an SVC routine by inserting another program in the path between the SVC instruction and the SVC routine. The inserted program is sometimes called a ***front end*** to the original SVC routine. After the front end program is inserted, the resulting body of code, including the front end program and the original SVC routine, is the new SVC routine.

Intercepting SVC routines can be recursive. Thus, if an SVC routine already has a front end, you can still add another front end onto it, and so on, indefinitely.

To intercept an SVC routine, you must obtain and save the address of the existing SVC routine for use by the front end program. To change the entry in the SVC table so it points to the front end program, you must use the REPLACE function of the SVCUPDTE macro.

In a user environment where the interception of SVC routines is recursive, it might be necessary to serialize the modification of the SVC table. To serialize, use the ENQ and DEQ macros to secure and hold the SYSZSVC TABLE resource while you are changing the SVC table.

Before you obtain the SVC table entry, use ENQ to secure this resource, and hold it until you have replaced the SVC table entry with the pointer to the front end routine. Then you can DEQ the resource. The major and minor names of this resource are, respectively, SYSZSVC and TABLE.

When securing the resource using the ENQ macro, specify the E parameter (to indicate an exclusive ENQ) if the resource is modified while under control of the task, and a scope of SYSTEM.

Extended SVC routines

SVC 109 provides you with extended SVCs that are available for you to define. Extended SVCs with function codes 0-199 are reserved for IBM. You may define only extended SVC with function codes 200-255. SVC 109 assumes the characteristics of a type 3 or 4 SVC. You can use the SVCUPDTE macro to specify those attributes your extended SVC routine requires, such as: locks required, restriction to APF authorized callers only, non-preemptable, and whether the SVC can be called from AR mode. Place the extended SVC routine in the LPA list and name the routine IGX00*yyy* where *yyy* is the number (200 - 255), in character format, that is loaded into register 15.

To execute an extended SVC routine:

- Load register 15 with a function code (200-255), and
- Issue SVC 109.

Subsystem SVC screening

After you write an SVC routine and insert it into the system, the routine is generally available unless you take steps to regulate access to the routine. Subsystem SVC screening allows a system routine to define those SVCs that a specific task can validly issue. When SVC screening is active for a task, the system determines, for each SVC issued by that task, whether the task can request that SVC function. If the SVC request is invalid, control is given to a special error subroutine supplied by the routine that activated the screening function.

The subsystem, executing under PSW key zero, activates SVC screening by setting two fields in each TCB for which screening is desired. The two fields consist of a screen flag bit and a one-word field containing the address of the subsystem screen table, which provides the interface between the SVC FLIH and the subsystem subroutine. In addition to these fields, the subsystem may optionally set the TCBSVCSP bit to indicate that ATTACH processing is to pass the SVC screening information to the attached task. The important SVC screening fields in the TCB are:

- TCBSVCS - A flag bit. When set to one, it indicates that screening is in effect for this task.
- TCBSVCA2 - Address of the subsystem screen table.
- TCBSVCSP - Propagation bit. When set to one, it indicates that ATTACH processing should pass the SVC screening information in these three fields to the attached task.

When the screening facility detects an invalid SVC, it gives control to the specified error routine. The error routine receives control as an SVC and is subject to the same restrictions as SVC routines. Before giving control to the subroutine, the SVC FLIH provides the setup for the subroutine as defined by the subsystem SVC entry (SSTSVCN) in the subsystem screen table. This setup includes:

- Initializing the SVRB if the subroutine is to execute as a type 2, 3, or 4 SVC.
- Obtaining the LOCAL lock if the subroutine is to execute as a type 1 SVC.
- Acquiring all locks necessary for the subroutine's execution.

The subsystem that needs SVC screening obtains storage via GETMAIN for an area called the subsystem screen table. For non-extended SVC screening, the subsystem screen table is 264 bytes in size. For extended SVC screening, the subsystem screen table is 1288 bytes in size. To prevent a page fault, this area must come from the LSQA (subpool 253-255), the SQA (subpool 245), or must be in fixed storage. If the subsystem screen table is in fixed storage, the subsystem must ensure that the storage is protected from user modification. The subsystem screen table contains two areas as follows:

1) SSTSVCN -- Subsystem SVC entry (8 bytes)

Byte	Bit	Contents
0-3	0	One of the following: • 0 - Indicates 24-bit addressing mode • 1 - Indicates 31-bit addressing mode
	1-31	Entry point address of the subsystem subroutine that will get control whenever a task has issued an SVC against which there is a screening restriction.
Byte 4		**X'00'** The subroutine is to run as a Type 1 SVC **X'08'** The subroutine may be used only by a program that is APF authorized **X'80'** The subroutine is to execute as a Type 2 SVC **X'C0'** The subroutine is to execute as a Type 3 or Type 4 SVC **X'20'** The subroutine is to execute as a Type 6 SVC
Byte 5		One of the following: • Bit 0 is used to indicate if the SVC may be issued in AR mode. Set bit 0 to 1 to indicate that the SVC may be issued in AR mode. • Bit 7 is used to screen individual extended SVCs. Set bit 7 to 1 to screen individual extended SVCs. When screening extended SVCs, the SVC screening mask must be 1280 bytes in size instead of 256 bytes.
Bytes 6-7		Locks to be held on entry to the subroutine. If the appropriate lock bit is one, the lock will be acquired by the SVC FLIH. The lock bits are: **Bit** **Lock** **0** LOCAL **1** CMS Bits 2–15 are always zero (off).

2) SSTMASK -- SVC screening mask (256 bytes or 1280 bytes)

If you are not screening extended SVCs (bit 7 of byte 6 of the SSTSVCN is zero), then SSTMASK will be 256 bytes long. If you are screening extended SVCs (bit 7 of byte 6 of the SSTSVCN is one), then SSTMASK must be 1280 bytes long.

Bytes **Content**

8-263 Each byte corresponds to an SVC number in ascending order in the range 0-255. When the high order bit in a byte is one, the task may validly issue the respective SVC; when the bit is zero, there is a screening restriction that prohibits the task from issuing the SVC.

264-519
Each byte corresponds to an SVC routing number in ascending order in the range 0-255 for SVC 109.

520-775
Each byte corresponds to an SVC routing number in ascending order in the range 0-255 for SVC 116.

776-1031
Each byte corresponds to an SVC routing number in ascending order in the range 0-255 for SVC 122.

1032-1287
Each byte corresponds to an SVC routing number in ascending order in the range 0-255 for SVC 137.

Note: Each SVC's data must be 256 bytes long, even though the system only supports all 256 routing codes for extended SVC 109.

When you use SVC screening with the ESR function, and a program issues an extended SVC, the system ignores the screening information provided for the SVC number itself (for example, 109). Instead, it looks only at the screening information provided for the particular routing code. See SVCUPDTE in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for more information about specifying the ESR parameter.

Note: Before terminating, the subsystem must clear the TCBSVCS bit, in all of the effected TCBs, before freeing the storage used for the screening table. Failure to turn off SVC screening by clearing the TCBSVCS bit can result in the loss of one or more systems. It is also recommended that the TCBSVCA2 screening table pointer be zeroed, and that the TCBSVCSP bit be cleared.

Chapter 24. Accessing unit control blocks (UCBs)

Each device in a configuration is represented by a unit control block (UCB). Primarily, you use UCB macros to:

- Scan for UCBs with certain characteristics or certain device numbers and receive a copy of each UCB or the addresses of its segments.
- Obtain the addresses of UCB segments.

Scanning for UCBs

There are two methods to scan for UCBs: the UCBSCAN macro and the UCB scan service. IBM recommends that you use the UCBSCAN macro rather than the UCB scan service.

The UCBSCAN macro is the general method for scanning UCBs. You can use it to scan for any UCB, including UCBs for dynamic devices, UCBs with 4-digit device numbers, and UCBs that reside above 16 megabytes. See “Obtaining UCB information (general methods)” on page 511 for more information about the UCBSCAN macro.

The UCB scan service is an older, limited method. It scans only for UCBs for static or installation-static devices, UCBs with 3-digit device numbers, and UCBs residing below 16 megabytes. The limited UCB scan service is described in “Obtaining UCB information (limited method)” on page 514.

Obtaining UCB addresses

You can obtain the address of a UCB segment for a given UCB common segment address or a given device number. There are several services that you can use to obtain addresses of UCB segments.

Figure 64 shows UCB segments. Information about the segments and how to access them follows.

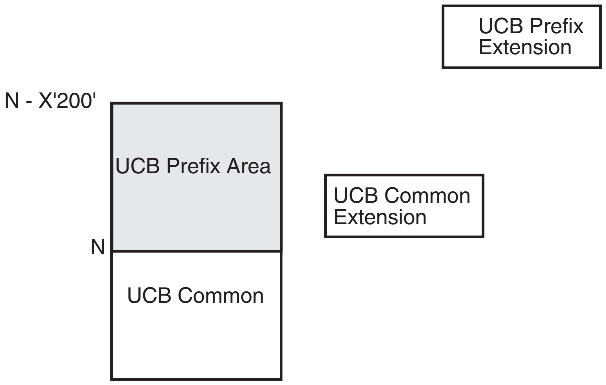


Figure 64. UCB Segments

UCB Common Segment

This segment resides below or above 16 megabytes depending on the UCB location that you specify for the HCD device definition. You can obtain its address with the following services:

- UCBLOOK
- UCBSCAN

Use the IEFUCBOB mapping macro to map the UCB common segment.

UCB Common Extension

This segment resides below or above 16 megabytes depending on the UCB location that you specify for the HCD device definition. You can obtain its address with the following services:

- IOSCMXA
- IOSCMXR
- UCBLOOK UCBCXPTR
- UCBSCAN UCBCXPTR

Use the UCBCMEXT DSECT of the IEFUCBOB mapping macro to map the UCB common extension.

UCB Prefix Extension

This segment always resides above 16 megabytes. You obtain its address with the following services:

- IOSUPFA
- IOSUPFR
- UCBLOOK UCBPXPTR
- UCBSCAN UCBPXPTR

Use the IOSDUPFX mapping macro to map the UCB prefix extension.

UCB Prefix Area

NOT Programming Interface information

This segment resides below or above 16 megabytes depending on the UCB location that you specify for the HCD device definition. The UCB prefix area is always the X'200' bytes preceding the UCB common segment. Therefore, if a UCB common segment is at address N, you can base the UCB on address N-X'200'. Use the IEFUCBOB mapping macro to map the UCB prefix area.

End of NOT Programming Interface information

UCB details

There are two methods to obtain a UCB segment address for a given device number: the UCBLOOK and IOSLOOK macros. IBM recommends that you use the UCBLOOK rather than IOSLOOK macro.

The UCBLOOK macro is the general method to obtain a UCB segment address for a given device. You can obtain any UCB address, including UCBs for dynamic devices, UCBs with 4-digit device numbers, and UCBs which reside above 16 megabytes. See "Obtaining UCB information (general methods)" on page 511 for more information about UCBLOOK.

The IOSLOOK macro is an older, limited method to obtain a UCB segment for a given device. It obtains addresses only for static or installation-static devices with 3-digit device numbers and with UCBs residing below 16 megabytes.

Although the IOSCMXA, IOSCMXR, IOSUPFA, and IOSUPFR macros might provide better performance than the UCBLOOK macro, other factors need to be considered. Table 67 can help you determine which of the macros for obtaining the addresses of UCB segments is most appropriate for your application.

Table 67. Comparison of macros that return addresses of UCB segments

Feature	**UCBLOOK**	**IOSCMXA** **IOSUPFA**	**IOSCMXR** **IOSUPFR** **IOSDCXR**
Function	Obtains the address of the following segments for a given device number: • UCB common segment • UCB common extension • UCB prefix extension Pins the UCB with the PIN parameter.	IOSCMXA — Obtains the address of UCB common extension for a given UCB common segment address. Note that the captured address is obtained for a given captured UCB common segment address. IOSUPFA — Obtains the address of UCB prefix extension	IOSCMXR — Obtains the address of UCB common extension for a UCB common segment address. Note that the captured address is obtained for a given captured UCB common segment address. IOSUPFR — Obtains the address of UCB prefix extension IOSDCXR — Obtains the address of UCB device class extension for a UCB common segment address. Note that the captured address is obtained for a given captured UCB common segment address.
Parameter Handling	In a parameter list	In a parameter list	Not in a parameter list The addresses are passed in general purpose registers (GPRs).
Recovery	Provided by macro	Caller must provide	Caller must provide
Environment	Can be issued in primary or access register (AR) ASC mode	Must be issued in primary ASC mode	Must be issued in primary ASC mode

The IOSCMXA, IOSCMXR, IOSDCXR, IOSUPFA, and IOSUPFR macros are described in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

You can use the UCBINFO macro to obtain device information from a UCB. For example, you can obtain information about device pathing, information about alias UCBs for a parallel access volume, and reasons why a device is offline. The UCBINFO macro is described in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

The following topics precede the description of the general methods for obtaining UCB information:

- "Ensuring that UCBs are not deleted" on page 498
- "Requesting notification of I/O configuration changes" on page 501

- “Detecting I/O configuration changes” on page 507
- “Retrieving the current MIH time interval” on page 508
- “Retrieving information about I/O hardware on an I/O path” on page 508
- “Validating I/O paths” on page 510
- “Obtaining device information for an allocation request” on page 510
- “Accessing above 16-megabyte UCBs” on page 518

These topics provide basic information to help you develop application programs that are to run in an environment where dynamic I/O configuration is used.

Ensuring that UCBs are not deleted

In a dynamic configuration environment, any program that obtains a UCB address must ensure that the UCB will not be deleted before the program has finished referencing the UCB. Pinning prevents a device's UCB and other related data structures from being deleted. Pinning a UCB ensures that the returned address for that UCB is valid as of the moment the system returned the address. It also ensures that the address remains valid as long as the UCB is pinned. You can pin a UCB by specifying the PIN parameter on the UCBLOOK macro, the UCBPIN macro, or the UCBSCAN ADDRESS macro. When the PIN parameter is specified on any of these macros, the system returns a pin token. The pin token identifies the pin, and will be required for unpinning the UCB.

Programs that pin a UCB are also responsible for unpinning it once the UCB is no longer being used. Unpinning allows a device's UCB and other related data structures to be deleted. Unpinning is done through the UCBPIN macro.

For a description of the circumstances under which pinning is not required, see “When pinning is not required” on page 501.

Pinning and unpinning UCBs

You can pin a UCB in either of the following ways:

- When issuing the UCBLOOK and UCBSCAN macros, specify the PIN parameter on these macros.

  Specifying the PIN parameter when obtaining UCB information with UCBLOOK and UCBSCAN ensures that the UCB cannot be deleted through a dynamic configuration change.
- Issue the UCBPIN macro, specifying the PIN parameter.

  Specifying the PIN parameter with UCBPIN ensures that the UCB whose address is supplied as input cannot be deleted through dynamic configuration changes. For example, assume that your program has allocated a device and must save the UCB address for later use after the device is unallocated. You could issue UCBPIN with the PIN parameter, providing the address of the UCB for the device as input. Pinning the UCB ensures that the UCB (and therefore the device itself) cannot be deleted once the device is unallocated.

You can unpin UCBs by issuing the UCBPIN macro with the UNPIN option; you must provide the pin token that the system returned when the UCB was pinned.

Programs that pin UCBs must provide information documenting the reason for the pin request. If a configuration change involving a pinned UCB is attempted, this information is displayed as part of a message that explains why the configuration change was rejected. Use the TEXT parameter to provide the text for the message. Potential text might record what is using the device, for example:

- THE XYZ SERVICE IS USING THE ABC DEVICE.

The system will reject any requested I/O configuration changes involving a device whose UCB is pinned. Therefore, you might want a program to receive notification of configuration change requests that involve a device whose UCB has been pinned by the program. The program could then unpin the UCB, allowing the configuration change to occur. See "Requesting notification of I/O configuration changes" on page 501 for further information.

Each program that needs to obtain UCB information must ensure that the UCB will not be deleted. Thus, any UCB might be pinned by several different programs at one time. Each program must subsequently unpin the UCB before the device can be deleted. For example, if a UCB has been pinned by three different programs and is later unpinned by two of those programs, the UCB is still pinned because of the one outstanding pin request.

If a program fails to unpin a UCB that it had previously pinned, the system will automatically free the pin at the time of termination of the task or address space with which the pin is associated, unless the LASTING parameter was specified when the UCB was pinned. The following list identifies four environments in which the system will automatically free an outstanding pin:

- A program that is in task mode and not in cross memory mode has pinned a UCB. The pin request is associated with the current job step task. If the pin is still in force when the job step task terminates, the system automatically frees the pin.
- A program that is in SRB mode and not in cross memory mode has pinned a UCB. The pin request is associated with the currently-executing address space. If the pin is still in force when the address space terminates, the system automatically frees the pin.
- A program that is in task mode and cross memory mode has pinned a UCB. The pin request is associated with the current job step task of the current home address space. If the pin is still in force when the job step task terminates, the system automatically frees the pin.
- A program that is in SRB mode and cross memory mode has pinned a UCB. The pin request is associated with the home address space. If the pin is still in force when the address space terminates, the system automatically frees the pin.

If LASTING is specified on the pin request, the system will not automatically free the pin.

When pinning is required

A UCB must be pinned in the following environments:

- The device represented by the UCB is offline and unallocated, but a program might still require access to the UCB and its related control block structures.

  Note: For a dynamic device to be deleted, it must be offline and unallocated, and its UCB must not be pinned. When a device is in the offline and unallocated state, the only way for a program to prohibit the deletion of the device is to pin the UCB for the device.
- The device represented by the UCB can become offline or unallocated while the program is accessing it.

- The UCB address is passed between programs that are running asynchronously.

Example: Pinning an unallocated and offline device

A program receives a device number for a DASD as input. The program is to format the specified device, and needs to ensure that the device cannot be deleted while it is being formatted. To accomplish these things the program must do the following:

- Pin the device by issuing the UCBLOOK macro with the PIN parameter, specifying the device number as input. UCBLOOK pins the UCB, and returns the UCB address and a pin token to the program.
- Format the DASD.
- Unpin the UCB through the UCBPIN macro with the UNPIN option, using the pin token returned when the UCB was pinned.

Example: Passing an UCB address between asynchronously running programs

A program dynamically allocates a device. Once the device is successfully allocated, it is not eligible for deletion. After the device is allocated, the program finds the UCB associated with the device. Because the program has allocated the device, there is no need to pin it.

However, if the program passes the address to another program or unit of work, the passing program must pin the UCB. Then the passing program can pass the UCB address and the pin token to another unit of work (such as a task) for its use. For example, a program stores the UCB address in a commonly addressable storage location and posts another task. Before issuing the POST macro, the program must pin the UCB with the UCBPIN macro and specify the LASTING parameter, and then pass the UCB address and pin token to the task. The task must then unpin the UCB after it has finished using the UCB.

Note: If you do not issue UCBPIN with LASTING prior to the POST, the following can occur:

1. The first program places the UCB address in a commonly addressable storage location and issues the POST macro without preceding it with a UCBPIN LASTING.
2. The program terminates after issuing the POST macro. The system frees the pin at task or address space termination because you did not specify LASTING on UCBPIN. The device whose UCB address the program passed is deallocated. (Assume that there are no other outstanding allocations against the device).
3. Before the posted work unit begins execution, the installation varies the device offline and then activates a new I/O configuration definition that does not include the device.
4. After the activation, the posted program begins execution and obtains the passed UCB address and attempts to reference it. In this instance, the posted task references invalid storage because the UCB is deleted after the device is deleted.

If the UCB had been pinned with UCBPIN LASTING by the first program before the POST macro was issued, the device could not have been deleted when the first program unallocated the device.

When pinning is not required

UCB pinning is not required if the program accessing the device can guarantee that the device will not be deleted while the program is accessing the UCB. A program can guarantee that the device (and thus the UCB) will not be deleted if:

- The environment is one in which dynamic I/O configuration changes cannot occur.
- The device cannot become unallocated and enter the offline state while the work unit is accessing the control structures that represent the device.
- The program receives a UCB address for the device as input, but does not invoke any asynchronous processing requiring that UCB address.

  Note: In this case, the calling program is responsible for pinning or allocating the UCB.

Example: Allocating a device

A device can be allocated to a program either through a JCL allocation or through the dynamic allocation interface. Once the device is allocated, it cannot be dynamically deleted. As long as the program keeps the device allocated, it can access the corresponding UCB without pinning. If the program needs to unallocate the device but still maintain the UCB address in its control structures for later use, it must first pin the UCB for the device through the UCBPIN macro before it unallocates the device.

Requesting notification of I/O configuration changes

If you have code that monitors or works with software I/O configuration, you can code authorized programs that track dynamic I/O configuration changes, and respond to requested and completed configuration changes.

You can use either the ENFREQ macro or the CONFCHG macro to request notification of I/O configuration changes. The ENFREQ macro is recommended because it is a general interface that allows authorized programs to listen for different types of system events. In contrast, CONFCHG can only be used to listen for I/O configuration changes.

Authorized programs can request the system to notify them of the following events:

- A requested or rejected I/O configuration change that involves deleting either a device or a path to a device (event code=31 on the ENFREQ macro, or CHGREQ parameter on the CONFCHG macro). A program that pins UCBs might need to know when a configuration change is requested that involves a device whose UCB the program has pinned. The program can then unpin the UCB to allow the requested change to be made. If the request is subsequently rejected, the program can re-pin the device.

  Note: See “Pinning and unpinning UCBs” on page 498 for more information on pinning and unpinning.
- A successful configuration change (event code=32 on the ENFREQ macro, or CHGCOMPL parameter on the CONFCHG macro). Programs might need to know when a change completed successfully so they can keep track of devices that are added, deleted or modified.

A program that requests notification of dynamic I/O configuration changes must be authorized, unlocked, enabled, and in task mode.

To request notification of dynamic I/O configuration changes in your installation, do the following:

- Code configuration change exit routines that will receive control for a particular notification. (See "Coding a configuration change user exit routine.") Code a different configuration change exit routine for each type of notification.
- Issue the ENFREQ or CONFCHG macro within your program for each notification you request. You must issue the macro separately for each type of notification.

The subtopics of this topic deal with how to use CONFCHG to request notification of I/O configuration changes. See Chapter 8, "Listening for system events," on page 159 for how to use ENFREQ to establish a listen request and code a listener user exit routine.

Using the CONFCHG macro

You must code the following parameters on CONFCHG to request notification of dynamic I/O configuration changes in your installation:

- The NOTIFY parameter to indicate that you want to be notified of I/O configuration changes
- The CHGREQ parameter or the CHGCOMPL parameter:
  - Code the CHGREQ parameter if you want to be notified of requested or rejected I/O configuration changes that involve deleting a device or deleting a path to a device.
  - Code the CHGCOMPL parameter if you want to be notified of I/O configuration changes that completed successfully.
- The EXIT parameter to identify the configuration change user exit routine that is to receive control
- The TOKEN parameter to cancel the notification request

If you no longer want to be notified of changes, you must cancel any notification requests by issuing CONFCHG CANCEL. If you plan to cancel notification requests, you must code the TOKEN parameter when you issue CONFCHG NOTIFY. The system then returns a token that you use when you code CONFCHG CANCEL. If your program terminates, you must cancel the notification request. If recovery is done through percolation, you must cancel each notification request separately.

See *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* for information on how to code the CONFCHG macro.

Coding a configuration change user exit routine

When an authorized program wants to be notified by the system about dynamic I/O configuration changes, the program identifies a configuration change exit routine on the CONFCHG macro (EXIT parameter). The configuration change exit routine pins, unpins, or tracks devices as required by the program.

Exit routine environment

The configuration change exit routine receives control in the following environment:

Environmental factor	**Requirement**
Authorization:	Supervisor state and PSW key 0.
Dispatchable unit mode:	Task

Environmental factor	**Requirement**
Cross memory mode:	PASN=HASN=SASN
AMODE:	31-bit
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held

Programming considerations

Consider the following when coding your configuration change exit routine:

- The exit routine is required to save and restore the contents of GPR 13.
- The exit routine must reside in common storage.
- The exit routine should not wait for asynchronous work to complete. Otherwise, system deadlocks might occur.
- Code a different configuration change exit routine for each type of notification, rather than coding one exit routine to respond to all notification types.

Entry specifications

The system passes information to the configuration change exit routine in a parameter list and in registers.

Registers at entry: On entry to the configuration change exit routine, the general purpose registers (GPRs) contain the following information:

Register	Contents
GPR 0	Undefined
GPR 1	Address of a pointer to the exit parameter list
GPRs 2-13	Undefined
GPR 14	Return address
GPR 15	Entry point address of the configuration change exit

Parameter list contents: The parameter list that the system passes to the configuration change exit routine is mapped by the IOSDDCCD mapping macro, and its address is pointed to by GPR 1. The parameter list includes information about the change that occurred, such as the specific devices being added, modified, or deleted.

For complete field names and lengths, offsets, and description of the fields mapped by the IOSDDCCD mapping macro, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Return specifications

On return to the system, the configuration change exit routine does not have to set any return codes or place any information in the GPRs. The configuration change exit routine must restore the contents of GPR 13.

Exit recovery

If the exit routine terminates abnormally, and the exit provides a recovery routine, the system passes control to the exit's recovery routine. If the exit does not provide a recovery routine, the system invokes its own recovery to terminate the notification request.

Exit routine processing

When an ACTIVATE command is issued, the system ensures that the devices to be deleted are offline and unallocated. If the activate request has passed this validation step, and an authorized program issues CONFCHG CHGREQ, the system passes control to the exit routine. When a requested activation change that involves deleting a device or deleting a path to a device is rejected, the system also passes control to the exit routine.

If the program issues CONFCHG CHGCOMPL, the system passes control to the exit routine when a dynamic I/O configuration change completes successfully.

You are responsible for writing the configuration change exit routine. This routine takes actions based on the needs of your program, and can do such things as pinning and unpinning devices, and keeping track of devices that are added, modified, or deleted.

Example: The following example illustrates a request to delete several devices, and explains the interaction between the installation program that requests notification of I/O configuration changes, the configuration change exit routine that receives control, and the system:

- An operator or system programmer wants to delete several devices, and activates an installation's I/O definition file (IODF) that causes the deletion of the devices.
- The system ensures that the devices to be deleted are offline and unallocated. If the request passes this validation step, the system passes control to the configuration change exit that the program specified on the CONFCHG CHGREQ macro (and to any other configuration change exit routines specified by other installation programs).

  Note: If the request does not pass this initial validation, the system rejects the change and issues notification messages to the operator.
- On entry to the exit routine, GPR 1 contains the address of a parameter list, which contains an indication of the change that was requested, and information about the devices being deleted. The parameter list data structure (the DCCD) is mapped by the IOSDDCCD mapping macro, described in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). To distinguish between a configuration change that was requested and a configuration change that was rejected, the exit routine can check the DCCDFUNC field in the parameter list. DCCDFUNC=DCCDFPFC (where DCCDFPFC is a system-defined constant) when this is a request for a configuration change.
- Depending on your installation needs, the configuration change exit might unpin the UCBs that were pinned by installation programs, so that the new configuration definition can take effect. To unpin the UCBs, the exit must issue the UCBPIN macro.
- After all the configuration change exits complete, the system determines if all the UCBs for devices to be deleted have been unpinned by all the exit routines. If this validation step succeeds, and if the configuration change completes successfully, the system passes control to the configuration change exit routine that the program specified on the CONFCHG CHGCOMPL macro (and to any other configuration change exit routines specified by other installation programs).
- The exit routine can then take the appropriate action. For example:

  - The routine might be keeping a table of all tape devices. Based on the information in the parameter list, the routine might then update its table.
  - The routine might pin UCBs for devices that were being added to the system.
- If any of the UCBs for the devices to be deleted have not been unpinned, the system rejects the entire configuration change. The system passes control to the configuration change exit routine the program specified on the CONFCHG CHGREQ macro. The system passes the parameter list to the exit routine with DCCDFUNC=DCCDFCCR (where DCCDFCCR is a system-defined constant), indicating the requested change was rejected. In this case, the exit routines might want to re-pin any UCBs they have unpinned, because the deletion will not take place. If the exit routines had deleted the devices from any tables they maintain, they might then add the devices back into the tables.

Coded example: CONFCHG macro invocation of configuration change user exit

In this example of using the CONFCHG macro and user exit, the exit resides in common storage and comes from an authorized library.

```
INVOCATION OF USER EXIT:

         MODESET MODE=SUP                    Enter supervisor state

⋮
         LA    R03,CHGEXIT                   Address of notify exit
         O     R03,HIBIT_ON                  Make sure high bit on
         ST    R03,EXIT_ADD                  Save exit address
*
*    Issue the CONFCHG macro so that CHGEXIT gets control
*    when a configuration change completes
*
         CONFCHG NOTIFY,CHGCOMPL,EXIT=EXIT_ADD,TOKEN=TOKEN,            X
               MF=(E,CONFPARM)
         MODESET MODE=PROB                   Return to problem state

CANCELLATION OF USER EXIT INVOCATION:
*    Use CONFCHG to indicate CHGEXIT should not be called
*    after configuration changes
*
         MODESET MODE=SUP                    Enter supervisor state

⋮
         CONFCHG CANCEL,CHGCOMPL,TOKEN=TOKEN,MF=(E,CONFPARM)
         MODESET MODE=PROB                   Return to problem state
CONFIGURATION CHANGE USER EXIT:

CHGEXIT  DS    0H
         BAKR  R14,0                    Save regs on link stack
         LR    R12,R15                  Set up base reg
         USING CHGEXIT,R12
         L     R03,0(R01)               Save DCCD address
         LA    R02,DYNLEN(0)            Dynamic area length
         STORAGE OBTAIN,LENGTH=(R02),ADDR=(R09),SP=0
         USING DYNAREA,R09              Get addressability to dyn area
         LA    R13,SAVEAREA             Get addressability to save area
         MVC   SAVEAREA+4(4),FIRSTSAV   First save area in chain
         USING DCCD,R03
*
*    Get addressability to DCCD entry array
*
         LR    R04,R03
         A     R04,DCCDSTRT             Reg 4 gets address of 1st entry
         USING DCCDARRY,R04
```

```
*
*    Calculate end address of DCCD
*
         LR   R05,R03
         A    R05,DCCDSIZE              Reg 5 gets address of 1 byte
                                        passed end of DCCD
*
*    Process each array entry
*    Use UCBLOOK to get the UCB address for each added UCB
*
LOOP     DS    0H
         CLI   DCCDETYP,DCCDDEV              Device entry ?
         BNZ   NEXT                          Branch if no
         CLI   DCCDEREQ,DCCDDADD             Add device entry ?
         BNZ   NEXT                          Branch if no
         TM    DCCDEFLG,DCCDESFT             Software entry ?
         BNO   NEXT                          Branch if no
*
*    Use UCBLOOK to find UCB address of added device:
*
         UCBLOOK DEVN=DCCDDEVN,UCBPTR=ADD_UCB,NOPIN,DYNAMIC=YES,       X
               RANGE=ALL,MF=(E,LOOKP)
*
*    Process the added UCB
*
NEXT     DS    0H
         LA    R04,DCCDELEN(R04)              Next entry
         CR    R04,R05
         BL    LOOP
*
*    Prepare to return to caller
*
         DROP  R09
*
*    Free the dynamic area
*
         LA    R02,DYNLEN(0)
         STORAGE RELEASE,LENGTH=(R02),ADDR=(R09),SP=0
         PR                                  Return to caller
         EJECT
*
*   Register equates
*
R01      EQU   1
R02      EQU   2
R03      EQU   3
R04      EQU   4
R05      EQU   5
R06      EQU   6
R07      EQU   7
R08      EQU   8
R09      EQU   9
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
*
*   Static area declares
*
         DS    0F
FIRSTSAV DC    CL4'F1SA'                     First save area ID
HIBIT_ON DC    X'80000000'                   Mask to turn on high bit
*
*   Dynamic area declares
*
DYNLEN   EQU   DEND-DYNAREA                  Dynamic area length
```

```
DYNAREA  DSECT
SAVEAREA DS    18F
TOKEN    DS    F                              Token from CONFCHG NOTIFY
ADD_UCB  DS    F
EXIT_ADD DS    F
*
*   Declare CONFCHG parameter list
*
         CONFCHG MF=(L,CONFPARM)
*
*   Declare UCBLOOK parameter list
*
         UCBLOOK MF=(L,LOOKP)
DEND     DS    0H
         IOSDDCCD
         END
```

Detecting I/O configuration changes

You can use the MVS I/O configuration token to detect I/O configuration changes. The MVS I/O configuration token is a 48-byte token that uniquely identifies an I/O configuration to the system. The token will change whenever the software configuration definition changes. Thus, if your program obtains the current I/O configuration token and compares it to one previously obtained, the program can determine whether there has been a change in the I/O configuration: If the tokens do not match, the I/O configuration has changed.

Both the MVS I/O configuration token and the ENFREQ macro (or the CONFCHG macro) with its related exit can be used to detect I/O configuration changes. (See "Requesting notification of I/O configuration changes" on page 501 for information on the ENFREQ or CONFCHG macro and related exit routine.) However, there are two important differences:

- The ENFREQ macro (or the CONFCHG macro) and related exit are available only to authorized programs, whereas both authorized and unauthorized programs can use the MVS I/O configuration token.
- With the ENFREQ macro (or the CONFCHG macro) and related exit, you can also receive notification of requested configuration changes that involve deleting a device or deleting a path to a device but have not yet occurred. With the MVS I/O configuration token, you can only detect changes that have already taken place.

An optional parameter, IOCTOKEN, is available with the UCB macros. Specifying IOCTOKEN ensures that the system will notify the caller through a return code and will not return any data if the current I/O configuration is not consistent with the configuration represented by the token that was specified as input by the caller.

There are two ways to obtain the current I/O configuration token:

- Issue the IOCINFO macro. See *z/OS MVS Programming: Assembler Services Reference IAR-XCT* for information on how to code the IOCINFO macro.
- Issue any of the UCB macros, setting the input specified by the IOCTOKEN parameter to binary zeroes. The macro will then return the current I/O configuration token.

  Note: For the UCBSCAN macro, setting the IOCTOKEN to binary zeroes returns the I/O configuration token at the start of the scan.

Use of the I/O configuration token can help prevent data inconsistencies that might occur if the I/O configuration changes between the time the caller obtained the token and the time the service returns the information. For example, you can use the configuration token to identify a case in which the I/O configuration changes during a UCB scan. If the IOCTOKEN parameter is specified with UCBSCAN, the system will notify the caller through a return code if the set of UCBs changes while the scan is in progress. Checking for the return code would allow the caller to restart the scan to ensure that all UCBs in the current configuration are referenced.

Specifying IOCTOKEN also allows the caller to receive notification if a dynamic device reconfiguration (DDR) swap occurs. DDR SWAP causes a token change.

The I/O configuration token can also serve to identify that the relationship between a device number and a UCB address has changed. This could be useful when a device number or UCB address is saved along with a configuration token and used for later processing. For example, a program may obtain a device number from the EDTINFO macro. The program may then want to issue UCBLOOK to obtain the UCB address for that device number. To ensure that the device definition of the UCB address obtained from UCBLOOK is consistent with the definition of the device number obtained by EDTINFO, the configuration token can be used.

In some cases, a change in the relationship between device number and UCB can only be detected through the use of an I/O configuration token. For example:

- A dynamic configuration change occurs that deletes a UCB and adds a different UCB at the same device number.
- A dynamic configuration change occurs that deletes a UCB. Then, a subsequent dynamic configuration change adds a new UCB at the same address as that of the deleted one.
- A dynamic device reconfiguration (DDR) swap occurs that swaps the contents of two UCBs.

Retrieving the current MIH time interval

When an I/O operation completes, an I/O interrupt occurs to signal the end of the operation. The MIH time interval is the period of time that the missing interrupt handler (MIH) waits before reporting that an expected interrupt from an I/O device is missing. At system initialization, MIH uses control statements in SYS1.PARMLIB to assign MIH time intervals to devices. If there is a missing interrupt, the system tries to resolve it and make the device usable again. It might clear the subchannel, end the operation, or try the operation again.

To retrieve the current MIH time interval for a device, use the MIHQUERY macro. This macro requires you to specify the address of the UCB or a copy of the UCB for the device. To determine the address of the UCB for a device, see “Obtaining UCB information (general methods)” on page 511.

Retrieving information about I/O hardware on an I/O path

By using the IOSCDR macro, you can retrieve information that uniquely identifies I/O hardware located along a specific I/O path. For example, you can obtain the model number and serial number, which you can use to uniquely identify a device across multiple systems.

The information that IOSCDR retrieves is returned in an area called a configuration data record (CDR), which is mapped by the mapping macro IHACDR. This CDR, sometimes called the self-description record, is generated by the device's control unit. If you try to obtain these records on devices that do not support self-description and you have specified READ=IO on the IOSCDR macro, you will receive a return code indicating that the device does not support the channel command words (CCWs) needed to generate the records.

The format of the IHACDR is documented in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). For more information about the contents of CDRs, see *ESA/390 Common I/O Device Commands*.

When you are coding the IOSCDR macro, you must consider the following:

- Length of the area that receives the CDR
- How IOSCDR retrieves the CDR
- The time that IOSCDR takes to perform I/O

Length of the CDR area

When coding IOSCDR, you must ensure that the CDR area is large enough to receive the entire CDR. If the length you specify on the CDRLEN parameter is smaller than the size returned in the CDRSIZE parameter, IOSCDR retrieves a partial CDR. As shown in Figure 65, the size of the actual CDR is 256 bytes, but CDRAREA is only 128 bytes. In this case, CDRAREA could hold only half of the actual CDR. To obtain the entire CDR, reissue IOSCDR with a CDRLEN value that is equal to or greater than the 256 bytes returned in CDRSIZE.

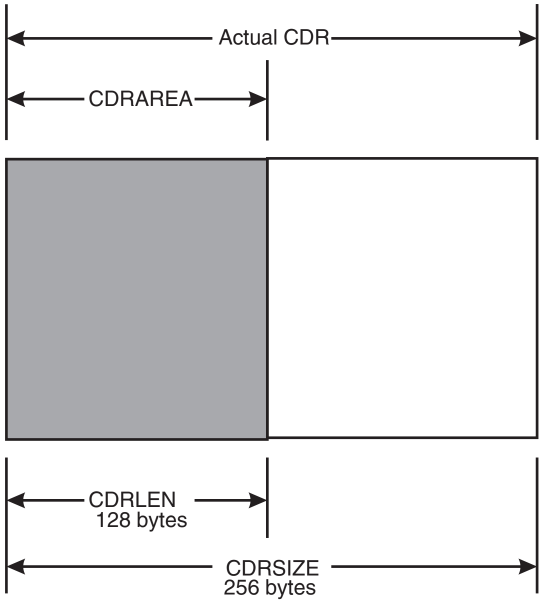


Figure 65. Comparison of CDRLEN to CDRSIZE

How IOSCDR retrieves the CDR

There are three ways that IOSCDR can retrieve a CDR:

- Directly from the specified device. Specified as READ=IO, this option is slower because it requires I/O, but it provides more current information than

READ=NOIO. This option will always give you the CDR that the control unit is currently providing when you issue IOSCDR.

- From the system. IOSCDR retrieves the last CDR known to MVS for the specified device and path. Specified as READ=NOIO, this option is quicker than READ=I/O, but it is not as current.
- Either from the system or directly from the device, depending upon whether the device and path are online. Specified as READ=COND, this option allows the system to decide how to retrieve the CDR. READ=COND returns the most accurate CDR in the shortest time possible, because it performs no I/O if the device and path are online.

Time that IOSCDR performs I/O

When coding the IOSCDR macro, you must also decide how long you want the macro to perform I/O before being purged. IOSCDR runs until one of the following occurs:

- IOSCDR completes, successfully or unsuccessfully
- The interval that you specify on the TIME parameter expires
- The MIH interval for the device expires.

Note that the TIME parameter allows you to set an expiration time that is specific to IOSCDR. The MIH interval, however, is used by all services associated with the device.

Validating I/O paths

The IOSPTHV macro enables authorized callers to determine if a channel path to a device is valid without changing the online/offline status of a path. It recognizes a channel path as valid if it is available. A path is considered available if an I/O operation can be initiated down a path, and the device can be selected. Validation does not, however, guarantee that the device and path are error free, because intermittent errors can exist.

You can find more specific information about the IOSPTHV macro in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Obtaining device information for an allocation request

The IEFDDSRV macro enables authorized callers to find the devices associated with an allocation request. For example, for a JCL input statement that requests the allocation of one or more data sets, you can use IEFDDSRV to obtain the addresses of the UCBs for the devices containing those data sets. When you specify that an above 16 megabyte UCB not be captured during dynamic allocation, you must use the IEFDDSRV macro to retrieve the UCB address.

You can find more specific information about the IEFDDSRV macro in *z/OS MVS Programming: Assembler Services Reference IAR-XCT*

Configuring a channel path online or offline

Use the IEEMCF macro to configure a channel path online or offline, or to determine the physical status of the channel path.

Before using IEEMCF, you must do the following:

- Decide how many channel paths you would like to reconfigure or query.

- Obtain a contiguous area of storage from subpool 247. Specify an area whose length is equivalent to 36 bytes multiplied by the number of channel paths that you are reconfiguring.
- Include the mapping macro IEEMRCFP.

To configure a channel path or determine its physical status, you must issue IEEMCF twice: once with TYPE=BUILD and once with TYPE=INVOKE. TYPE=BUILD builds the parameter list containing information about the devices to be reconfigured or queried. TYPE=INVOKE configures or queries the physical status of the devices in the list built by TYPE=BUILD and receives data and return codes from the system.

Obtaining UCB information (general methods)

Use the UCB macros described in this topic to obtain UCB information. These macros can be used for any UCB, including those defined as dynamic (dynamic UCBs).

Obtaining UCB addresses for a specified device number

You can use the UCBLOOK macro to obtain actual (not captured) addresses of the following segments for a UCB:

- UCB common segment
- UCB common extension segment
- UCB prefix extension segment

The input UCBs can be associated with:

- A device with a 3-digit hexadecimal number
- A device with a 4-digit hexadecimal number
- For DASD and tape devices, the volume serial number

IBM recommends using the UCBLOOK macro to obtain UCB addresses.

The PIN parameter on the UCBLOOK macro allows the caller to pin the UCB whose address is to be obtained. See “Pinning and unpinning UCBs” on page 498 for more information on pinning and unpinning.

The IOSLOOK macro is restricted to below 16 megabyte UCBs for static or installation-static devices with 3-digit device numbers. Like UCBLOOK, IOSLOOK returns the UCB address associated with a given device number. However, UCBLOOK has several advantages that IOSLOOK does not:

- UCBLOOK can be used for dynamic devices, as well as static and installation-static devices. UCBLOOK provides the option to limit the look-up to static and installation-static devices, or to also include dynamic devices.
- UCBLOOK can be used for devices with 4-digit device numbers as well as for devices with 3-digit device numbers.
- UCBLOOK can be used for above 16 megabyte UCBs as well as below 16 megabyte UCBs.
- UCBLOOK allows an EBCDIC device number or, for DASD and tape devices, a volume serial number, to be used as a look-up argument.
- UCBLOOK supports AR-mode callers.

Scanning UCBs

Use the UCBSCAN macro to scan UCBs, returning UCB information on each invocation. The scan can include all UCBs in the system, or be restricted to a specific device class. For example, you can use UCBSCAN to find all DASD devices currently defined in the I/O configuration.

There are two kinds of information that UCBSCAN can return on each invocation. If you specify the COPY keyword, UCBSCAN returns a copy of a UCB in a user-supplied work area. If you specify the ADDRESS keyword, UCBSCAN returns a UCB common segment address. With the ADDRESS keyword, you can also request a UCB common extension or prefix extension address. Unauthorized programs must specify the COPY keyword when invoking the UCBSCAN macro.

When you use UCBSCAN to obtain UCB addresses for devices, it is necessary to ensure that these devices will not be dynamically deleted while the scan is taking place. The PIN parameter on the UCBSCAN macro allows the caller to pin each UCB whose address is to be obtained. As with UCBLOOK, specifying the PIN parameter ensures that the returned address for the UCB is valid as of the moment the UCB is pinned, and that the address remains valid as long as the pin is in effect. As in any other case where a program pins a UCB, the program is responsible for unpinning the UCB once the UCB is no longer subject to processing.

IBM recommends the use of the UCBSCAN macro for UCB scans. The limited method UCB scan service (described under “Obtaining UCB information (limited method)” on page 514) is restricted to below 16 megabyte UCBs for static and installation-static devices with 3-digit device numbers. However, the UCBSCAN macro can perform the same functions as the UCB scan service, and also provides these additional advantages:

- You can use UCBSCAN for dynamic devices, as well as for static and installation-static devices. It is also possible to limit the scan to static and installation-static devices, or to also include dynamic devices.
- You can use UCBSCAN for devices with 4-digit numbers as well as for devices with 3-digit numbers.
- You can use UCBSCAN COPY for copies of below and above 16 megabyte UCBs.
- You can use UCBSCAN ADDRESS for addresses of below and above 16 megabyte UCBs.
- You can use UCBSCAN ADDRESS to obtain the address of the UCB common extension and UCB prefix extension.
- UCBSCAN supports AR-mode callers.
- UCBSCAN allows you to start the scan at a specific device number.
- UCBSCAN returns UCBs in ascending order by device number.

Examples: Using the UCB macros

Example 1

Obtain the UCB address for the device whose volume serial number is specified by VOL. The look-up should include UCBs for dynamic devices. Pin the UCB, and return the UCB address at the address specified by UCBPTR. After all processing related to the UCB has completed, unpin the UCB.

```
         UCBLOOK VOLSER=VOL,UCBPTR=UCBPTR,DYNAMIC=YES,PIN,             X
               TEXT=TEXTLOOK,PTOKEN=PTOKLOOK
*
*    Process UCB
*
         UCBPIN UNPIN,PTOKEN=PTOKLOOK
*
*        ...
*
UCBPTR   DS   F
VOL      DS   CL6                          VOLSER searching for
TEXTLOOK DC   CL58'PIN TEXT FOR UCBLOOK'     Pin Text
PTOKLOOK DS   CL8                          Pin Token
```

Example 2

Obtain the addresses of the UCBs for all tape devices. The scan should include UCBs for dynamic devices. On each invocation of UCBSCAN, return a UCB address at the address specified by UCBPTR and pin the UCB whose address is returned. After all processing related to each UCB has completed, unpin the UCB.

```
*
*        .....
*
SETUP    DS    0H
         XC    SCANWORK,SCANWORK             Clear work area
SEARCH   DS    0H
         UCBSCAN ADDRESS,UCBPTR=UCBPTR,WORKAREA=SCANWORK,              X
               DEVCLASS=TAPE,DYNAMIC=YES,PIN,TEXT=TEXTSCAN,            X
               PTOKEN=PTOKSCAN
         LTR   R15,R15                       Has a UCB been returned?
         BNZ   DONE                          No  end of device class
*
*    Process  UCB
*
         UCBPIN UNPIN,PTOKEN=PTOKSCAN
         B     SEARCH
DONE     DS    0H
*
*        .....
*
UCBPTR   DS    F                             UCB copy from SCAN
PTOKSCAN DS    CL8                           Pin Token
SCANWORK DS    CL100                         Work area for UCBSCAN
TEXTSCAN DC    CL58'PIN TEXT FOR UCBSCAN'    Pin Text
```

Determining if the UCB macros (general methods) are available

Programs that need to determine if the UCB macros are installed can use one of the following procedures. These procedures are provided for programs that can also run on versions of MVS which do not support the UCB macros.

Note: The UCB macros (general methods) are supported on MVS/SP Version 4 and above, as well as on MVS/SP Version 3 systems if the dynamic I/O configuration compatibility APARs are installed.

Procedure for pre-MVS/SP Version 4 libraries

Use this determination method when the source code is compiled against pre-MVS/SP Version 4 macro libraries. (Note: The UCB macros must be included in the macro library concatenation.)

```
            L    10,X'10'               Load CVT pointer
            USING CVT,10
            TM   CVTDCB,X'08'           Is the OSLEVEL extension present?
```

```
         BNO  INVOKE_LIM          No, pre-MVS/SP Version 3 system
         TM   X'4F0'(10),X'08'    General methods supported?
         BNO  INVOKE_LIM          No, general methods unavailable
         UCBLOOK ........         Invoke general method
         B    BYPASS_LIM          Branch around limited method
INVOKE_LIM  DS 0H
         IOSLOOK                  Invoke limited method
BYPASS_LIM  DS 0H
```

Procedure for MVS/SP Version 4 and above libraries

Use this determination method when the source code is compiled against MVS/SP Version 4 and above macro libraries.

```
         L    10,X'10'            Load CVT pointer
         USING CVT,10
         TM   CVTDCB,CVTOSEXT     Is the OSLEVEL extension present?
         BNO  INVOKE_LIM          No, pre-MVS/SP Version 3 system
         TM   CVTOSLV0,CVTUCBSV   General methods supported?
         BNO  INVOKE_LIM          No, general methods unavailable
         UCBLOOK ........         Invoke general method
         B    BYPASS_LIM          Branch around limited method
INVOKE_LIM  DS 0H
         IOSLOOK                  Invoke limited method
BYPASS_LIM  DS 0H
```

Obtaining UCB information (limited method)

Note: IBM recommends that you use the UCBSCAN macro rather than the UCB scan service.

The UCB scan service described in this topic can be used only for UCBs that have not been defined as dynamic. See “Scanning UCBs” on page 512 for information on performing scans that include UCBs defined as dynamic.

The UCB scan service

The UCB scan service allows you to scan each UCB in the system or in a specified device class. The device classes are: tape, communication, channel-to-channel adapter, direct access, display, unit record, and character reader. Using UCB scan services you can, for example, find the UCB currently associated with a particular VOLSER or find all tape devices currently defined.

The UCB scan service runs in the caller's key, state, and addressing mode. The caller can be in either task or SRB mode; if in task mode, the caller must be enabled and hold no locks. Unlike the IOSLOOK macro, the UCB scan service does not require the user to be in supervisor state or to provide the device number as input.

Invoking the UCB scan service

Each time that you invoke the UCB scan service, you will obtain the address of the common segment of one UCB. To scan several UCBs, you must invoke the UCB scan service repeatedly, once for each UCB. The UCB scan service keeps track of your position in the UCB chain by information that it stores in the 100-byte work area that you provide as input. To start your scan, clear this work area to binary zeroes. The zeroes indicate that the UCB scan service is to start the scan at the first UCB in the system or device class. If you want to continue the scan to obtain the next UCB, do not change the work area.

When you have obtained the address of a UCB common segment, you can use the UCBDEVN macro to convert the device number referred to by the UCB into a four-byte printable EBCDIC form.

Input to the UCB scan service

To use the UCB scan service, the caller must:

- Obtain a 100-byte work area that starts on a doubleword boundary. To start with the first UCB in the system or the first UCB in a device class, clear the work area to binary zero. To continue scanning UCBs, do not change the work area between calls to the scan service routine.
- Build a parameter list.

Parameter List for the UCB Scan Routine:

- Address of the 100-byte work area provided by the caller.
- Address of the byte containing the device class to which the search is being restricted. See the topic “Limiting the UCB Scan” for information on how to restrict the search to a specific device. If all UCBs are to be scanned, the byte pointed to must contain X'00'.
- Address of the word in which the UCB scan service is to return the UCB address. The high order bit of this field must be 1 to indicate it is the last word in the parameter list.

Set up the registers to contain the following information:

Register	**Contents**
1	Address of the parameter list
13	Address of caller's 18-word save area
14	Caller's return address
15	Entry point of the UCB scan routine. (The CVTUCBSC field in the CVT contains the entry point address.)

Note: The data areas that the caller passes to the UCB scan service must be addressable in the addressing mode of the caller. If the program runs in 31-bit addressing mode, the data areas can be anywhere; otherwise, the data areas must be below 16 megabytes.

Limiting the UCB scan

If you want to limit the UCB scan to a specific device class, you must provide the address of a one-byte field containing the hexadecimal code for that class. These fields are defined in the UCBDVCLS (or UCBTBYT3) bit string in the UCB. Figure 66 on page 516 lists the valid device class specifications with their UCB definitions. For example, to restrict the search to tapes, set the byte containing the device class equal to the constant UCB3TAPE. If you use the UCB definitions in your program, you must include the UCB mapping macro (IEFUCBOB). To scan all of the UCBs in the system, provide the address of a one-byte field containing X‘00’.

UCB Definition
Device Class

UCB3TAPE
Tape

UCB3COMM
Communication

UCB3CTC
Channel-to-channel adapter

UCB3DACC
Direct access

UCB3DISP
Display

UCB3UREC
Unit record

UCB3CHAR
Character reader

Figure 66. Device Classes

Output from the UCB scan service

When the UCB scan service returns, register l5 contains one of the following return codes:

Return Code
Meaning

00 The UCB scan service stored a UCB address in the location specified in the third word of the parameter list.

04 There are no more UCBs. The UCB scan service set the 100-byte work area to binary zeros.

Note:

1. A dynamic device reconfiguration (DDR) swap might occur during a scan. Because this type of swap results in the interchange of information in UCBs, it might cause a UCB address to be skipped or returned twice.
2. Do not place any dependencies on the order in which the UCB addresses appear during a scan. The address of the UCB representing device 250, for example, might be returned before the one representing device 140.
3. Devices with optional channels are associated with only one UCB. Therefore, the UCB scan service returns only one UCB address for those devices. Devices with multiple exposures have one UCB associated with each exposure. Therefore, the UCB scan service returns one UCB address for each exposure.

Example: Using the UCB scan service

The following is an example of how to use the UCB scan service to find the UCB currently associated with a particular VOLSER. The search is limited to direct access UCBs.

```
FINDVOL      CSECT
PROLOG       STM      14,R12,12(R13)
             BALR     R12,0
PSTART       DS       OH
             USING    PSTART,R12
             ST       R13,MYSAVE+4              SAVE CALLER'S REGISTER 13
             LA       R2,MYSAVE                 GET MY SAVE AREA ADDRESS
             ST       R2,8(R13)                 CHAIN SAVE AREA TO CALLER'S
             LR       R13,R2                    SET UP TO USE LOCAL SAVE AREA
             ......
             ......
SETUP        DS       OH                        SET UP FOR UCB SCAN SERVICE
             L        R3,CVTPTR                 GET CVT ADDRESS
             USING    CVTMAP,R3                 SET UP ADDRESSABILITY TO CVT
             XC       WORKAREA,WORKAREA         CLEAR WORK AREA
             LA       R1,WORKAREA               GET ADDRESS OF WORK AREA
             ST       R1,PARMWA                 STORE ADDRESS IN THE PARMLIST
             LA       R1,DEVCLASS               GET ADDRESS OF AREA CONTAINING           X
                                                THE DEVICE CLASS TO BE SEARCHED
             ST       R1,PARMDEVT               STORE ADDRESS IN THE PARMLIST
             MVI      DEVCLASS,UCB3DACC         INDICATE ONLY DIRECT ACCESS UCBS
*                                               ARE TO BE SEARCHED.
*                                               NOTE:  IF ALL UCBS WERE TO BE
*                                                      SEARCHED, DEVCLASS WOULD
*                                                      BE SET TO X'00'.
             LA       R1,ADDRUCB                GET ADDRESS OF WORD WHERE SCAN           X
                                                SERVICE WILL STORE THE UCB ADDRESS
             ST       R1,PARMUCB                STORE ADDRESS IN THE PARMLIST
             OI       PARMUCB,X'80'             INDICATE END OF PARMLIST
             USING    UCBOB,R2                  SET UP ADDRESSABILITY TO UCB
SEARCH       DS       OH
             LA       R1,PARMLIST               PUT PARMLIST ADDRESS IN REGISTER 1
             L        R15,CVTUCBSC              GET SCAN SERVICE ADDRESS
             BALR     R14,R15                   GO TO SCAN SERVICE.
*                                               INTERFACE:
*                                                 REGISTER 1 =  ADDRESS OF THE
*                                                               PARAMETER LIST
*                                                 REGISTER 13=  ADDRESS OF AN
*                                                               18-WORD SAVE AREA
*                                                 REGISTER 14=  RETURN ADDRESS
*                                                 REGISTER 15=  SCAN SERVICE ENTRY
*                                                               POINT ADDRESS
             LTR      R15,R15                   HAS A UCB BEEN RETURNED?
             BNZ      NOMATCH                   NO, AT END OF DEVICE CLASS AND           X
                                                NO MATCH FOUND
             L        R2,ADDRUCB                GET UCB ADDRESS THAT THE SCAN            X
                                                SERVICE RETURNED

             CLC      UCBVOLI,SRCHVOL           IS THIS THE VOLSER WE'RE LOOKING         X
                                                FOR?
             BNE      SEARCH                    NO, CONTINUE SCAN OF UCBS.
*                                               NOTE:  THE WORK AREA MUST NOT BE
*                                                      CHANGED BETWEEN CALLS TO THE
*                                                      SCAN SERVICE ROUTINE
FOUND        DS       OH
             ......
             ......
NOMATCH      DS       OH
             ......
             ......
ENDIT        DS       OH
             L        R13,MYSAVE+4              RESTORE CALLER'S REGISTER 13
             LM       R14,R12,12(R13)           RESTORE REMAINDER OF CALLER'S            X
                                                REGISTERS
             BR       R14
             EJECT
PARMLIST     DS       3F                        PARMLIST MAPPING
             ORG      PARMLIST
PARMWA       DS       F                         ADDRESS OF 100-BYTE WORK AREA
PARMDEVT     DS       F                         ADDRESS OF BYTE CONTAINING               X
                                                THE DEVICE TYPE TO BE SEARCHED
PARMUCB      DS       F                         ADDRESS OF WORD TO CONTAIN THE           X
```

```
                                                UCB ADDRESS
             SPACE
DEVCLASS     DS       CL1                       BYTE CONTAINING DEVICE CLASS TO          X
                                                BE SEARCHED FOR
ADDRUCB      DS       F                         WORD IN WHICH UCB SCAN WILL PLACE        X
                                                THE ADDRESS OF THE UCBS.  ALIGN
             DS       0D                        ON DOUBLE-WORD BOUNDARY.  (THE
*                                               WORK AREA FOR SCAN SERVICE MUST
*                                               BE ON A DOUBLE-WORD BOUNDARY.)
WORKAREA     DS       CL100                     WORK AREA
MYSAVE       DS       18F
             ......
             ......
             DSECT
             IEFUCBOB                           UCB MACRO ID
             CVT      DSECT=YES
EJECT
END          FINDVOL
```

Obtaining the subchannel number for a unit control block (UCB)

The IOSINFO macro obtains the subchannel number for a specified UCB from the system without being dependent on the location or format of the information as it is maintained in the UCB. The macro returns the subsystem identification word (SID), which identifies the subchannel number of the UCB, in a user-specified location. The SID is a fullword value; it contains the subchannel number in its ending halfword. (The first halfword contains X'0001'.)

IOSINFO obtains the number of the subchannel that was associated with the UCB at NIP time. However, the subchannel and the UCB might become disassociated during system operation. Any disassociation of the UCB and the subchannel means the subchannel number in the SID might not be valid. Therefore, IOSINFO returns information consistent with NIP time but does not guarantee that the subchannel will always be associated with the UCB.

If the UCB is disassociated from the subchannel at the time of the IOSINFO macro invocation, IOSINFO can detect the situation and notify the user through a return code. If the UCB is disassociated from the subchannel after the IOSINFO macro invocation, IOSINFO can not notify the caller. Instructions for coding the macro appear in *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Accessing above 16-megabyte UCBs

To conserve below 16-megabyte common virtual storage, you can define a device's UCB above 16 megabytes. You specify whether a UCB can exist above 16 megabytes in the hardware configuration definition (HCD) device definition. For application programs that require 24-bit addresses, the system enables above 16 megabyte UCBs to be accessed in below 16 megabyte private, virtual storage.

During allocation, the system automatically creates a below 16 megabyte view into the actual above 16 megabyte UCB. The view is known as a captured UCB. It enables an application to access the UCB in the private storage of its address space. The system automatically captures an above 16 megabyte UCB at allocation and releases the UCB at deallocation. With dynamic allocation, you can choose not to capture a UCB if affected applications can handle above 16 megabyte UCBs.

If you are not using standard IBM services, for example, you are building your own data extent block (DEB), you might need to control capturing of an above 16 megabyte UCB explicitly. For example, a program might allocate a device in one address space but perform I/O in another address space. A captured UCB created during allocation in the first address space is not accessible in the second address space. The program could explicitly capture the UCB in the second address space. You use the IOSCAPU macro to explicitly capture and release UCBs.

Although it is not recommended, you can use the CAPTCOM parameter on the IOSCAPU macro to capture a UCB in common rather than private storage. Capturing in common storage enables a UCB pointer in a common control block to be accessed in any address space. For this reason, the system automatically captures the UCBs for devices with the SYS1.LINKLIB, SYS1.SVCLIB, or logrec data sets in common rather than in private storage. To enable support for JES2 spool data sets, the UCBs for these devices are also captured in common storage. Therefore, if your common storage is constrained, the UCBs for these devices should remain below 16 megabyte UCBs.

Programs can also use the IOSCAPU macro to determine a captured UCB's actual UCB address. The IOSCAPF macro also provides a method of receiving an actual UCB address and handles parameter passing in general purpose registers (GPRs). IOSCAPF does not provide parameter validation or recovery.

With the UCBLOOK and UCBSCAN ADDRESS macros, you can use the LOC parameter to indicate whether you want to receive above 16 megabyte UCB addresses. The default is to receive only below 16 megabyte UCBs.

Macros that receive a UCB address accept the address of a below 16 megabyte UCB, above 16 megabyte UCB, or captured UCB. The UCBLOOK and UCBSCAN ADDRESS macros, which return UCB addresses, only return actual UCB addresses, not captured UCB addresses. The IOSUPFA and IOSUPFR macros also only return an actual, not captured, UCB address. Other macros that return addresses of UCB segments, such as the IOSCMXA, IOSCMXR, and IOSDCXR macros return the address of an actual UCB segment given an actual UCB address and the address of a captured UCB segment given a captured UCB address.

Chapter 25. Dynamic allocation

This information provides the following introductory information about dynamic allocation:

- An allocation overview
- Choosing the type of allocation for your program
- Dynamic allocation functions
  - In a batch environment
  - In an interactive environment
- Installation Options for Dynamic Allocation Functions.

If you have experience using dynamic allocation functions, you might want to skip this information and go on to Chapter 26, "Requesting dynamic allocation functions," on page 543, which describes the interfaces needed to code a dynamic allocation request.

If you are unfamiliar with the concept of allocation, read "An allocation overview," and all of the information that apply to your program.

To decide whether dynamic allocation is appropriate for your program, read "Choosing the type of allocation for your program" on page 522.

To learn which dynamic allocation functions are meaningful in your programming environment, read:

- "Using dynamic allocation functions in either a batch or interactive environment" on page 525, if your program is to run in a batch environment.
- "Using dynamic allocation functions in either a batch or interactive environment" on page 525 and "Using dynamic allocation functions in an interactive environment" on page 538, if your program is to run in an interactive environment, such as TSO/E. The functions described in both topics are useful in an interactive environment.

To find out what installation defaults can affect dynamic allocation, read "Installation options for DYNALLOC macro functions" on page 541.

Note:

BPXWDYN is a text interface to a subset of the SVC 99 (dynamic allocation) and SVC 109 (dynamic output) services. BPXWDYN supports data set allocation, unallocation, concatenation, and the addition and deletion of output descriptors. BPXWDYN is designed to be called from REXX, but it may be called from several other programming languages, including Assembler, C, and PL/I. For more information, see *z/OS Using REXX and z/OS UNIX System Services*.

An allocation overview

Allocation is the process by which the system assigns, or **allocates**, I/O resources to your job. An I/O resource is a ddname-data set combination, with any associated volumes and devices.

Deallocation is the process by which the system releases, or **deallocates**, I/O resources that were allocated to your job.

Choosing the type of allocation for your program

There are two basic types of allocation: job step allocation and dynamic allocation. The two types allocate resources at different points in program processing. Job step allocation assigns resources to your program before your program runs, and dynamic allocation assigns resources to your program while it is running. The needs of your program determine which type of allocation you should use.

Characteristics of job step allocation

When using job step allocation, you request I/O resources through JCL. The system allocates those I/O resources before your program runs, as part of initiating the job step, and deallocates resources after your program runs, as part of job step termination. This type of allocation ensures that the resources you request are available before your program runs, and throughout program execution.

Characteristics of dynamic allocation

When using dynamic allocation, you request I/O resources by coding the DYNALLOC macro and filling in the fields of the SVC 99 parameter list. The system allocates and deallocates those I/O resources while your program is running. Dynamic allocation also allows you to request information about your allocation environment, and to deallocate or modify characteristics of your allocation environment that were acquired either dynamically or through JCL.

Dynamic allocation allows you to tailor your device allocations based on input to your program. You can design your program to dynamically allocate only those devices that are necessary in a particular programming path, rather than allocating all possible device requirements before your program runs.

Dynamic allocation also allows you to use common resources more efficiently. When there is high contention for a resource, dynamic allocation allows you to acquire an I/O resource just before you need it and to release it just after you need it, so that your program holds the resource for a shorter length of time.

When to avoid using dynamic allocation

Do not use dynamic allocation in the following types of programs, because their characteristics, combined with a dynamic allocation request, can cause the program to end abnormally:

- Programs running in cross memory mode.

  For further information on the use of cross memory services, see *z/OS MVS Programming: Extended Addressability Guide*.
- Programs running under an interruption request block (IRB).

  Requesting dynamic allocation functions in routines that run under an IRB and issue OPEN, OPENJ, CLOSE, EOV, or FEOV, or any other service that enqueues on SYSZTIOT can cause a X'138' abend.
- Installation exits that get control during the start of the job entry subsystem.
- Installation exits that get control before batch allocation is complete.
- User exits for OPEN/CLOSE/EOV, or for any other routines that enqueue on SYSZTIOT.
- Multitasking programs in which one task issues the DYNALLOC macro with the S99TIONQ bit on in S99FLAG2, and other tasks issue the DYNALLOC, OPEN, OPENJ, CLOSE, EOV, FEOV, or LOCATE macros.

- MVS command installation exits.

  For further information about this restriction and about command installation exits, see *z/OS MVS Installation Exits*.

Additional requirements and restrictions for dynamic allocation are outlined in "Programming considerations for using the DYNALLOC macro."

Programming considerations for using the DYNALLOC macro

Before deciding to use any of the dynamic allocation functions, consider the environment of the program that invokes the DYNALLOC macro. Your program interacts with the job entry subsystem, with data management functions, and with system components that control the jobs, in addition to dynamic allocation itself.

The following list contains more specific programming considerations for using the DYNALLOC macro. Other considerations are included with the topics to which they apply.

- **Serialization of Resources**

  Your program might serialize the same resources as DYNALLOC. Dynamic allocation can serialize the following resources, depending on the path taken in processing.

Major Name	**Minor Name**
SYSDSN	data set name
SYSIEFSD	CHNGDEVS
SYSIEFSD	DDRDA
SYSIEFSD	DDRTPUR
SYSIEFSD	Q4
SYSZOPEN	data set name
SYSZPCCB	PCCB
SYSZTIOT	address of the DSAB QDB.asid
SYSZVMV	ucbaddr
SYSZVOLS	volume serial number

- **Other system routines and dynamic allocation**

  System routines invoked by various paths of dynamic allocation processing also might serialize a system resource. Some of the system functions invoked by dynamic allocation processing are LOCATE, OBTAIN, CATALOG, SCRATCH, and DADSM Allocate.

For the same reason, installation exits for OPEN/CLOSE/EOV, or for any other routines that enqueue on SYSZTIOT, should not issue dynamic allocation requests.

- **AMODE and RMODE considerations**

  There are no AMODE or RMODE restrictions for the caller of DYNALLOC. However, all addresses in the parameter list must be in 31-bit addressing mode format. 24-bit addresses must be 4 bytes long, with the high-order byte set to zero.

- **Avoiding 05C abends**

  Note:

  Programs that issue DYNALLOC should not receive control during START (initialization) processing for LOGONs, MOUNTs, or started tasks.

  Programs that get control during START processing (installation exits, for example) should not issue LOCATE, OPEN, OBTAIN, CATALOG, SCRATCH, or DADSM Allocate for data sets that have not been preallocated to the program; to do so will cause an 05C abend.

  Subsystems that receive control during step allocation as a result of the JCL SUBSYS parameter should not issue DYNALLOC; to do so might cause an 05C abend.

- **Changes to the TIOT**

  Dynamic allocation might cause changes to the task input/output table (TIOT). Depending on the function requested through the DYNALLOC macro, an entry could be added, deleted, or reordered; you cannot assume a fixed order for TIOT entries.

  You should make sure your program can handle changes to the TIOT. This is especially important when the EXTRACT macro is being used in your program, or in the program that will gain control when dynamic allocation processing is finished,

  If you need to reference TIOT entries after DYNALLOC is invoked, use the GETDSAB macro.

NOT Programming Interface information

- **Accessing ICF CATALOGS**

  Programs that get control during ‘START’ (installation exits, for example) should not issue LOCATE, OPEN, OBTAIN, CATALOG, SCRATCH, or DADSM allocate for data sets that have not been preallocated to the program; to do so could cause the system to issue message IEC331I with return code X'4' and reason code X'84'.

End of NOT Programming Interface information

Selecting the type of allocation based on program requirements

The table below recommends the type of device allocation to use based on the needs of your program.

When...	Choose:
You need data to be available for the duration of your program	JCL

When...	Choose:
You need to know that the program has access to all data before execution begins	JCL
Your data requirements are constant for all program conditions	JCL
You need data only in certain paths of program processing	Dynamic allocation
Your program can wait or fail if the data is not available when you issue the DYNALLOC macro	Dynamic allocation
Your program is a long-running job or server that could cause contention for system resources	Dynamic allocation

Dynamic allocation functions

Most dynamic allocation functions are useful in either a batch environment or an interactive environment, such as TSO/E. These functions are discussed in "Using dynamic allocation functions in either a batch or interactive environment." Some dynamic allocation functions are useful only in an interactive environment. These are described in "Using dynamic allocation functions in an interactive environment" on page 538.

If you are coding a program to run in a batch environment, read "Using dynamic allocation functions in either a batch or interactive environment."

If you are coding a program to run in an interactive environment, read both "Using dynamic allocation functions in either a batch or interactive environment" and "Using dynamic allocation functions in an interactive environment" on page 538 for a complete description of the dynamic allocation functions available to your program.

Using dynamic allocation functions in either a batch or interactive environment

The dynamic allocation functions described in Table 68 are useful in a batch environment or an interactive environment. They are:

Table 68. Dynamic Allocation Functions in a Batch or Interactive Environment

Function	Described in topic	Purpose
Dsname or pathname allocation	"Using dsname or pathname allocation" on page 526	Dynamically assigns a data set or z/OS UNIX file to a job by its data set name.
Ddname allocation	"Using ddname allocation" on page 540	Dynamically reuses a not-in-use data set.
Deallocation	"Deallocating resources" on page 533	Dynamically releases resources assigned to a job through JCL or dynamic allocation
Concatenation	"Concatenating resources" on page 536	Logically associates allocated data sets.
Deconcatenation	"Deconcatenating resources" on page 537	Logically disassociates concatenated data sets.

Table 68. Dynamic Allocation Functions in a Batch or Interactive Environment (continued)

Function	Described in topic	Purpose
Information retrieval	"Obtaining allocation environment information" on page 537	Retrieves information about your allocation environment.

Functions that are useful *only* in an interactive environment are described in "Using dynamic allocation functions in an interactive environment" on page 538.

Note: Throughout this information, the word "deallocate" is used to denote the action, and the word "unallocated" is used to denote the state. In cases where the common usage is hard to change – for example, in the name of a dynamic allocation function – without causing confusion, the words "unallocate/unallocation" have been retained.

Using dsname or pathname allocation

The major function performed by dynamic allocation, and the function most often requested, is that of dynamically allocating a data set or z/OS UNIX file according to its name (dsname or pathname). Dynamic allocation by dsname or pathname is equivalent to data set or file allocation during job step initiation, except that the resource is allocated as your program runs.

Before using dsname or pathname allocation, you should ensure that the service you need is available through dynamic allocation. You can request most of the JCL facilities that you can code in a DD statement – such as data set disposition, volume label information, expiration date, and SYSOUT destination – by specifying the appropriate text units in the parameter list. However, some JCL facilities do not have dynamic allocation equivalents. These facilities are described in "JCL DD statement facilities not supported by dynamic allocation" on page 527.

Consult the detailed description of each text unit key (see "SVC 99 parameter list verb codes and text units, by function" on page 625) for the capabilities supported by the key. The system might support a JCL subparameter but not support all values of that subparameter. For example, the system might support DCB=DSORG without supporting DCB=DSORG=IS.

After you have determined that you can use dsname allocation to fulfill your request, you might be concerned about doing it efficiently. You make the most efficient use of dsname allocation processing when you reuse existing allocations. Consider the following questions:

- Does an existing resource conflict with this request? (See "Checking for conflicts with your existing allocation environment" on page 527.)
- Can I use an existing resource for this request? (See "Using an existing allocation to fulfill a dsname allocation request" on page 528.)
- Can I modify an existing resource for this request? (See "Changing the parameters of an existing allocation" on page 530.)

If the answers to all the questions above are "no," the system uses a new allocation to satisfy your request, as described in "Using a new allocation to fulfill a dsname or pathname allocation request" on page 531. Allocating a new resource requires more processing than using an existing one, and so is less efficient.

JCL DD statement facilities not supported by dynamic allocation

Table 69 lists JCL DD statement facilities that *cannot* be used in dynamic allocation.

Table 69. JCL DD Statement Facilities Not Supported by Dynamic Allocation

Restricted DDnames	JOBCAT, STEPCAT, JOBLIB, and STEPLIB
Keyword Parameters	CHKPT, DDNAME, DLM, and DSID
Positional Parameters	*, DATA, and DYNAM
Selected Subparameters of Keywords	**Keyword** **Subparameter Not Supported** **DCB** reference to ddname of a previous step CYLOFL NTM RKP **DISP** PASS specification **DSN** reference to ddname (as in *.ddname) ISAM area name **SPACE** ABSTR specification **UNIT** AFF **VOLUME** RETAIN specification REF=ddname

Dynamic allocation facilities without JCL equivalents

Dsname allocation allows you to request the following data set characteristics, which are not available through JCL:

- The password for a password-protected data set. If you specify the password in your program through the SVC 99 parameter list, the system does not prompt the operator.
- The permanently allocated attribute.
- The convertible attribute.
- Return of certain information.

Checking for conflicts with your existing allocation environment

When you invoke the DYNALLOC macro to perform dsname dynamic allocation, an “allocation environment” already exists for your request. It consists of the allocation requests made through your JCL or earlier dynamic allocations, that have not yet been deallocated. The system considers these resources to be **existing allocations**, and goes to them first to fill your dynamic allocation requests.

Dynamic allocation cannot satisfy a dsname allocation request that is in conflict with your existing allocation environment. Environmental conflicts can cause your request to fail when your dsname allocation request specifies:

- A ddname that is associated with an existing allocation that is in use.
- A ddname that is associated with a group of concatenated data sets defined as permanently concatenated. (For a definition of permanently concatenated, see “Requesting the permanently concatenated attribute” on page 536.)

- A ddname that is associated with an existing allocation that does not have the convertible attribute or that does not fulfill the conditions listed under "Using an existing allocation to fulfill a dsname allocation request."
- A new non-temporary data set with the same dsname as that of an existing allocation.

  This is not a conflict if the request specifies a different volume serial number. Non-temporary data sets can have the same dsname if they do not reside on the same volume.
- An existing data set (by specifying a disposition of OLD or SHR) that is not permanently allocated, not in-use, and has a disposition of DELETE. A data set with these characteristics might be deleted before your program requests it.

  This is not a conflict if your request specifies a different volume serial number, and the allocation on the specified volume is permanently allocated and does not have a disposition of DELETE.

MVS might allocate a different data set than you intend when you have multiple versions of the same data set, with the same data set name, residing on different volumes. When a data set with the same name as the intended data set has been previously allocated (perhaps using an alias name) but not freed, then the current allocate request might not allocate the intended version.

Avoid allocation of the incorrect data set in the following ways:

- Specify the volume and unit where the data set to be allocated resides by using the DALVSER and DALUNIT text units in the dynamic allocation parameter list. See "Dsname allocation text units" on page 631 for more information.

  If the data set to be allocated is not cataloged in the master catalog, and a data set with the same name is cataloged in the master catalog, then you **must** specify DALVSER and DALUNIT.
- Free existing allocations for the same data set name (or alias names) using dynamic unallocation.
- Use a unique data set name.
- Set the no-conversion flag in the dynamic allocation block (S99NOCNV in S99FLAG1) to prevent the use of existing allocations for the same data set name. See "Setting up the request block" on page 547 for more information. An alternate method is to code the IEFDB401 installation exit to set the S99NOCNV flag. See *z/OS MVS Installation Exits* for more information.

Using an existing allocation to fulfill a dsname allocation request

If possible, dynamic allocation will use an existing allocation — an allocated resource marked not-in-use — to satisfy your dsname allocation request. Although some parameters can be changed if necessary, the request and the existing allocation must match according to several criteria before the allocation can be selected to satisfy your request.

Characteristics required in your request: To be satisfied by an existing allocation, your request must be for one of the following:

- An explicit data set name (dsname)
- The allocation of your terminal as an I/O device
- A dummy data set.

Characteristics prohibited in your request: To be satisfied by an existing allocation, your request must **not** specify any of the following:

- Data set sequence number.

- DCB reference.
- Label type.
- Parallel mounting.
- Pathname.
- Private volume.
- Unit count.
- Unit description. (If the dsname is in the form “&dsname”, the unit name description is ignored.)
- Volume count.
- Volume reference.
- Volume sequence number.
- Attributes represented by text units keys from X'8000' through X'BFFF'.

Characteristics required in the existing allocation: To be used to satisfy your request, the data set that is the existing allocation must have the following properties:

- It must not be in use.
- It must not be a member of a concatenated group.
- It must have the same volume serial number as any explicitly specified in the request.
- It must have the permanently allocated attribute, if its disposition is DELETE and the request specifies a status of MOD.
- It must not be a generation data group data set.
- It must not be a z/OS UNIX file.
- It must either have the convertible attribute (described in “Convertible attribute” on page 540) or, if the request is in a form other than “&dsname”, **all** of the following must be true:
  - The request does not specify a ddname; or the specified ddname matches the ddname associated with the existing allocation. A terminal request that does not specify a ddname cannot be satisfied by an existing allocation that does not have the convertible attribute.
  - For partitioned data sets, the member name specified in the request is the same as the member name associated with the existing allocation; or a member name is neither specified in the request nor associated with the existing allocation.
  - The request does not specify input only, output only, or any DCB parameters.
  - If the request specifies a status of MOD, MOD is also associated with the existing allocation; or it is neither specified in the request nor associated with the existing allocation.
  - The request does not specify that the convertible attribute be assigned to the allocation.
  - The request does not specify that only existing allocations with the convertible attribute may be used.

If the request specifies dsname in the form “&dsname”, the request does not specify a ddname; or the specified ddname matches the ddname associated with the existing allocation. A terminal request that does not specify a ddname cannot be satisfied by an existing allocation that does not have the convertible attribute.

System selection from multiple matching allocations: Even with all the restrictions listed here, more than one existing allocation could match your dsname request. Then, if you specified a ddname and one of the matching existing allocations is associated with that ddname, that is the allocation that dynamic allocation selects to satisfy your request.

If you did not specify a ddname, dynamic allocation selects the matching existing allocation whose in-use attribute was most recently removed. (Data sets allocated through JCL are considered to have had their in-use attributes removed at step allocation.)

System creation of new ddname for existing allocation: An existing allocation might not match your request even though it is associated with the same ddname you specify. Because the ddname is going to be associated with the resource that is allocated to your program, the system gives the existing allocation a new ddname, of the form 'SYS' followed by five digits. The association of a system-generated ddname with an existing allocation cannot occur when the existing allocation:

- Is in use
- Is open
- Does not have the convertible attribute
- Is associated with a permanently concatenated group that does not represent an entire generation data set group or a multi-device-type VSAM data set.

Changing the parameters of an existing allocation

When dynamic allocation uses an existing allocation to satisfy a dsname allocation request, some of the parameters of the existing allocation might have to be changed to match the parameters specified in the request. Only existing allocations that were dynamically allocated, with the convertible attribute, can have their parameters changed. (The convertible attribute is described in "Convertible attribute" on page 540.) Resources allocated through JCL or the TSO/E ALLOCATE command cannot have their parameters changed (with the exception of status and disposition specified through JCL), but they may be used if no changes are necessary.

The following parameters are eligible for change by dynamic allocation:

- Ddname
- Member name
- Status
- Normal disposition
- Conditional disposition
- Space
- Deallocation at CLOSE
- Input only
- Output only
- DCB attributes
- Password
- Permanently allocated attribute.

No other parameters may be changed.

Note:

1. You cannot change an exclusive status to shared status. For example, you cannot change OLD to SHR. However, it is possible to change SHR to OLD if no other jobs are enqueued on the requested data set.
2. You cannot change the parameters on an explicitly referenced OUTPUT JCL statement (DALOUTPT).
3. The status (DALSTATS) of an existing allocation to be reused/converted must always be SHR, OLD, or MOD. A Dynamic Allocation Information Retrieval request may return a status (DINRTSTA) of NEW for that existing allocation, because DINRTSTA will return the status as of the beginning of the step. DALSTATS must specify the status at the time that the dynamic allocation is being performed. See "Obtaining allocation environment information" on page 537 for more on requesting allocation status.

```
If a DD statement created the allocation environment as NEW:

      //OUTDATA  DD  DSN=TSI.TP31.DEV.OUTDATA.KAM,
      //             DISP=(NEW,DELETE),
      //             UNIT=SYSDA,SPACE=(TRK,(9,9),RLSE)

and you specify Information retrieval as:

       DINDDNAM  'OUTDATA '

and receive back information as:

       DINRTDSN              'TSI.TP31.DEV.OUTDATA.KAM'
       DINRTSTA              X'04' (NEW)       <==============
       DINRTNDP              X'04' (DELETE)
       DINRTCDP              X'00'
       DINRTATT              X'20' (Permanently allocated)
       DINRTTYP              X'00'

In order to convert the Normal Disposition from DELETE to CATLG
you must specify:

       DALDDNAM  'OUTDATA '
       DSLDSNAM  'TSI.TP31.DEV.OUTDATA.KAM
       DALSTATS  X'08' (OLD)       <======================
       DALNDISP  X'02' (CATLG)
       DALPERMA
```

Using a new allocation to fulfill a dsname or pathname allocation request

Dynamic allocation attempts a **new allocation** when it cannot satisfy your request with an existing allocation. New allocations cannot be processed by dynamic allocation while a job step holds (for possible reuse) more dynamically allocated resources than permitted by the control limit. See "Control limit" on page 539 for more information about the control limit, and "Permanently allocated attribute" on page 540 for an explanation of which data sets are eligible for automatic deallocation.

Considerations when requesting dsname or pathname allocation

The following topics describe how the system processes your dsname allocation request based on the attributes of your allocation environment and your request.

Allowing the system to generate a ddname: If you do not specify a ddname, the system generates one. The ddname created consists of the characters 'SYS' followed by five digits, with a maximum value of SYS65535.

Specifying a password: You may specify passwords as part of a dynamic allocation request to bypass prompting the operator.

Allocating a data set with a status of MOD: If you allocate a data set with a status of MOD but do not specify any volume information, and the data set cannot be found in the catalog, it is treated as a new data set.

Allocating a dummy z/OS UNIX file: If you specify both the DUMMY and pathname text units in the DYNALLOC macro parameter list, dynamic allocation processing treats the allocation as a dummy. You can also allocate a dummy file by specifying a pathname of **/dev/null** for the pathname text unit.

Determining the status of a z/OS UNIX file: Dynamic allocation determines whether the z/OS UNIX file to be allocated is NEW, OLD, or MOD from the pathname options specified on the DALPOPT text unit:

- The file is NEW if OEXCL and OCREAT are both specified.
- The file is MOD if OCREAT is specified but OEXCL is not specified.
- The file is OLD if OCREAT is *not* specified or if the DALPOPT text unit is not specified.

Cataloging a data set: If you specify a normal disposition of CATLG for a new direct access data set, the system catalogs the data set when it is allocated rather than when it is deallocated. If the data set cannot be cataloged, then no allocation will take place; if the data set cannot be allocated, it will not be cataloged.

Requesting a data set that is in use: Rather than wait for another user to release a data set, volume, or device to obtain use of it, dynamic allocation fails a request by an unauthorized program. If an authorized program specifically requests a wait, dynamic allocation will wait.

Retrieving information from allocation: You can request that the ddname, data set name, volume serial number, and other information assigned by allocation be returned in the DYNALLOC macro parameter list.

Retrieving data set organization information from allocation: You can also request that the data set organization (DSORG) of the allocated data set be returned in the DYNALLOC macro parameter list. Dynamic allocation returns whatever you specify as the DSORG, if anything. If you do not specify a DSORG on the allocation request, the system assigns and returns a data set organization according to the following defaults:

- If the allocation request is for a terminal as an I/O device or for a SYSOUT data set, 'PS' (physical sequential) is returned as a default value.
- If the allocation request is for a tape data set, 'PS' is returned as a default value.
- If the allocation request is for a NEW direct access data set, 'PO' (partitioned organization) is returned if you specified a non-zero directory space quantity; otherwise, the data set is assigned the DSORG of 'PS'. The 'PS' default value is not stored into the data set control block (DSCB) until the data set has been opened. If the newly allocated data set has not been unallocated, the default DSORG value can be obtained by issuing the information retrieval function. (See "Obtaining allocation environment information" on page 537 and "Dynamic information retrieval text units" on page 697.)
- If the allocation request is for an existing direct access data set, the data set organization obtained from the data set control block (DSCB) is returned. If the DSORG cannot be obtained from the DSCB, the allocation request fails.

- If the allocation request is for a non-existing non-SMS-managed data set, for which a status of SHR or OLD has been specified, a request for return of the DSORG causes the dynamic allocation to fail unless a DSORG value is also supplied on the request.
- For other types of allocation requests in which you do not specify a DSORG for the data set, the system returns zeros in the DYNALLOC macro parameter list field.

Access methods that are not available with dynamic allocation: You cannot create ISAM data sets through dynamic allocation.

You can create VSAM data sets through dynamic allocation only when the storage management subsystem (SMS) is active.

Requesting an allocated but unavailable data set: If you request an allocation by dsname and the dsname is already allocated but not available, dynamic allocation allocates the data set to the same unit and volume. "Using an existing allocation to fulfill a dsname allocation request" on page 528 describes the conditions under which an allocated data set will not be available to satisfy your request.

Allocating a GDG data set: An allocation of a GDG data set refers to the same data set for the life of the job (or TSO/E logon session), even if another generation is added during the job, unless the bit S99GDGNT is set in the FLAGS1 field of the request block. When this bit is on, the system issues a LOCATE to find the GDG base level. See "Flags (S99FLAG1)" on page 548 for a description of the bit, and Table 73 on page 549 for an example using S99GDGNT. See *z/OS MVS JCL User's Guide* for more information about GDG data sets.

Retrieving volume information: Dynamic allocation retrieves volume information from the volume serial number, a VOL=REF, or, if the data set is cataloged, from the catalog. Dynamic allocation will not use passed data set information to retrieve volume information.

Exceeding the control limit: The **control limit**, described on "Control limit" on page 539, limits the number of DD statements in a job step. Usually, the control limit is a concern only in an interactive environment. However, some utilities that can be used in a batch environment might require that you modify the control limit. Utilities such as IDCAMS expect a control limit that is larger than the number of DD statements in the job step. If the control limit is not larger, these utilities could receive an X'043C' error reason code from DYNALLOC (see Table 79 on page 606). If a batch program requires a large number of allocations, use the DYNAMNBR parameter on the EXEC statement to increase the control limit. Refer to *z/OS MVS JCL User's Guide* to choose a value for the DYNAMNBR parameter.

Deallocating resources

Deallocation releases resources allocated to your program. You can dynamically deallocate resources that were allocated either dynamically or through JCL, unless the data set is:

- Open
- A member of an open concatenated group
- A private catalog.

In any of these cases, the data set is not deallocated.

Data sets are unallocated:

- When you explicitly deallocate them
- When they are closed if FREE=CLOSE is specified
- At the end of the job step if not deallocated earlier

Note: A data set is not DEQed until it is unallocated by the last step whose JCL references it

When a data set is unallocated:

- You can use the ddname in subsequent dynamic allocation requests.
- The system processes the data set disposition.
- The system frees the unit to which the data set was allocated if it is not being used for any other DD within the step.
- The system releases the volumes on which the data set was allocated if it is not being used for any other DD within the step.
- The data set is DEQed and other jobs can use the data set, except when the:
  - System has deleted it as part of disposition processing (the data set will no longer exits and therefore cannot be referenced by any other jobs).
  - Data set is referenced in the JCL of a later step of this job (the data set is not DEQed until it is unallocated by the last step whose JCL references it).
- Although temporary data sets are never able to be used by other jobs, if the last reference to a temporary data set in the job is PASS, the system releases a:
  - Temporary VIO data set at the end of the last step in which it is referenced

    Note: The system never ENQs VIO data sets.
  - Releases a batch-allocated temporary non-VIO data set at the end of the job.

Deallocating resources by dsname or pathname

The following considerations apply to unallocation requests specifying a **dsname** or **pathname**. For a description of unallocation by **ddname**, see “Deallocating resources by ddname.”

- If you do not specify a ddname, and the dsname or pathname is associated with more than one ddname, all associated data sets or z/OS UNIX files are deallocated. If an error occurs while deallocating one ddname, processing continues for the others and an error code is returned in the SVC 99 parameter list. If errors occur for more than one ddname, the error code applies to the last ddname for which there was an error.
- If you specify a member name with the dsname, only those associations containing both the member name and dsname are deallocated. If you code a member name text unit key, you must also code the dsname text unit key for a valid request.
- To dynamically deallocate a z/OS UNIX file by pathname, you must specify the same pathname that you specified to allocate the file. If the pathname specified differs, the unallocation request will fail. For example, suppose that the file **pay.time** can be accessed through multiple paths. It was allocated as **/usr/applics/pay.time** but can also be accessed as **/comn/pgms/pay.time**. Deallocating the file through the **/comn/pgms** path is not possible. It must be deallocated by the **/usr/applics/pay.time** pathname.

Deallocating resources by ddname

The following considerations apply to deallocation requests specifying a **ddname**:

- Only the occurrence of the data set associated with the specified ddname is deallocated, even if that data set is associated with other ddnames.

- If a dsname, pathname, or dsname and member name, are specified in addition to the ddname, they must be associated with that ddname or the request fails.

Deallocating concatenated groups

If the specified resource is associated with a permanently concatenated group, described in “Requesting the permanently concatenated attribute” on page 536, the in-use attribute is removed from all members of the group, and the count of the number of resources held for reuse is increased by the number of members in the group. (See “In-use attribute” on page 538 for a description of the in-use attribute.)

An exception occurs when the concatenated group was generated by the system, such as generation data groups that are GDG ALL groups. In these cases, the group is treated as a single resource.

If a concatenated group has the permanently concatenated attribute and you specify a ddname with a dsname, or GDG ALL, the entire group is released. If you specify a dsname with GDG ALL, the request for dynamic unallocation fails. (See “Deallocating resources by ddname” on page 534 for considerations when deallocating resources by ddname.)

GDGs are described in *z/OS MVS JCL User's Guide*.

If the concatenated group does not have the permanently concatenated attribute, the group is deconcatenated and the member associated with the specified dsname is released. (The first member is released if the group's ddname is specified.)

Changing parameters at dynamic unallocation

With your dynamic unallocation request, you can include text units to change a data set's or z/OS UNIX file's parameters as it is being deallocated. If your request is in the form of verb code 02, key 8, the changes are honored when the data set or z/OS UNIX file is actually released, unless they have been overridden in the meantime. If your dynamic unallocation request is in the form of verb code 02, key 7, the changes take effect immediately.

The parameters that can be changed at deallocation are:

- Output class — see “Overriding SYSOUT class specification - Key = '0018'” on page 693
- HOLD/NOHOLD parameters — see “Overriding SYSOUT nohold specification - Key = '000A'” on page 692 and “Overriding SYSOUT hold queue specification - Key = '0059'” on page 693
- Remote work station destination — see “Overriding SYSOUT remote workstation specification - Key = '0058'” on page 693:
  - For a JES2 environment SYSOUT data can be routed to:
    - Remote workstation
    - Node
    - Special local
    - Userid
  - For a JES3 environment SYSOUT data can be routed to:
    - ANYLOCAL
    - JES3-defined device name
    - Device number
    - JES3-defined symbolic group name
    - NJE node

- Userid — see “Overriding SYSOUT user ID specification - Key = '0063'” on page 693

  Note: For a JES2 environment only.
- Data set or z/OS UNIX file disposition — see “z/OS UNIX overriding file disposition specification - Key = '801A'” on page 694
- When the SYSOUT output is to be printed (JCL DD statement SPIN parameter) — see “z/OS UNIX file path specification - Key = '8017'” on page 705.

Allocation disposition cannot be overridden for the following (see “Overriding disposition specification - Key = '0005'” on page 692):
- Passed data sets
- VSAM data sets, when SMS is not active
- System-named data sets, except for subsystem data sets.

For all other types of data sets, the disposition specified on an unallocation request overrides the disposition specified at allocation.

You cannot delete members of partitioned data sets with a disposition of DELETE; the system deletes the entire data set. An overriding disposition of DELETE for data sets allocated as shared is not valid; the system fails the overriding disposition request.

Concatenating resources

Dynamic concatenation logically connects allocated data sets into a concatenated group. You can identify data sets to be concatenated only by their associated ddnames. These data sets must not be open; if they are, the request for dynamic concatenation fails.

The order in which you specify the ddnames is the order in which the system concatenates their associated data sets. The name associated with the concatenated group is the ddname that was specified first; the other ddnames are no longer associated with any data set.

If a ddname you specify is associated already with a concatenated group, that entire group is included in the new concatenation.

After the request for dynamic concatenation is satisfied, all members of the dynamically concatenated group are assigned the in-use attribute. The in-use attribute is described on “In-use attribute” on page 538.

Requesting the permanently concatenated attribute

You can request that a concatenated group created through DYNALLOC be assigned the permanently concatenated attribute. A group with the permanently concatenated attribute has the following characteristics:
- The group cannot be dynamically deconcatenated into its member data sets.
- Except for the first data set in the concatenation, the data sets in the concatenation are no longer associated with their original ddnames. Instead, all data sets in the concatenation are associated with the ddname of the first data set in the concatenated group.
- If a permanently concatenated group is dynamically concatenated with other data sets to form a new non-permanently concatenated group, the permanently concatenated group remains intact if the new group is dynamically deconcatenated.

- If the group is not a system-defined permanently concatenated group, it is automatically assigned the permanently allocated attribute.

  Note: To dynamically release a non-system-defined permanently concatenated group, you specify the ddname, not the dsname, in the unallocation request.

A concatenated group defined through JCL is automatically assigned the permanently concatenated attribute, as is a concatenated group defined by the system through JCL or DYNALLOC. A GDG ALL request and a request for a VSAM data set that spans device types are examples of the latter situation.

Deconcatenating resources

Dynamic deconcatenation logically disconnects the members of a dynamically concatenated group. You identify the concatenated group to be deconcatenated by specifying the ddname of the group.

When a concatenated group is dynamically deconcatenated, the ddnames that were associated with the data sets before they were concatenated are restored unless this would result in duplicate ddnames. This situation could arise if a dynamic allocation with the ddname to be restored occurred after a dynamic concatenation. In this case, the deconcatenation request fails.

The request for dynamic deconcatenation also fails if the concatenated group is open.

Dynamic deconcatenation has no effect on the in-use attributes associated with the members of the group.

Obtaining allocation environment information

Dynamic information retrieval provides you with information about your current allocation environment. You can request allocation information using any of the following:

- The dynamic information retrieval function (verb code 07)
- The dsname allocation function
- The ddname allocation function.

The information returned is the information available at the time that the allocation information request is processed. In cases where the requested information is not applicable or not available, you receive 0 in the length field of the text unit, unless otherwise specified in the text unit description. For example, if you request that the system return the member name of a sequential data set, the system returns 0 in the length field, because sequential data sets do not have member names.

You can ask for information about any or all of your currently-allocated requests by specifying a relative request number. For example, you can obtain information about all your allocation requests by successively asking for information about the 1st, 2nd,...nth allocation request. Code the DINRTLST text unit key (key 13) with this series of requests, to receive an indication of the **last** relative entry.

Some of the information that you can request using verb code 07 includes:

- Data set name
- Ddname
- Pathname
- Member name

- Data set organization
- Status
- Normal disposition
- Abnormal disposition

Note: Dynamic information retrieval returns the compressed version of the z/OS UNIX pathname (multiple slashes are reduced to a single slash). Information retrieval will return a pathname of **/dev/null** for dummy z/OS UNIX files, regardless of the pathname originally specified on the JCL or dynamic allocation.

"Dynamic information retrieval text units" on page 697 describes the information that you can retrieve.

Using dynamic allocation functions in an interactive environment

Dynamic allocation provides some controls designed specifically for the interactive environment. The following features help to avoid tying up resources that are not being used, or allow you to reuse resources that have already been allocated.

Feature	Described in topic	Purpose
In-use attribute	"In-use attribute"	Determines whether a data set is eligible to be used by another dynamic allocation, or determines whether a data set or z/OS UNIX file can be deallocated if the control limit is reached.
Control limit	"Control limit" on page 539	Limits the number of data sets that can be allocated but not in use.
Permanently allocated attribute	"Permanently allocated attribute" on page 540	Prevents the system from automatically deallocating a data set to meet the control limit.
Convertible attribute	"Convertible attribute" on page 540	Allows dynamic allocation to change certain parameters of an existing allocation to match a dynamic allocation request.
Ddname allocation	"Using ddname allocation" on page 540	Allows you to reuse a previously allocated data set that is not in use.

In-use attribute

When a data set or a z/OS UNIX file is dynamically allocated, the system assigns it the in-use attribute. You can request that the system remove the in-use attribute by requesting that the system deallocate the data set or z/OS UNIX file, or by requesting the remove-in-use function. When the system marks a dynamically allocated data set or z/OS UNIX file as "not-in-use," it does not deallocate the resource:

- For a data set that has been marked "not-in-use", the data set becomes eligible for use in a subsequent dynamic allocation request.

- For an z/OS UNIX file that has been marked “not-in-use”, the file *cannot* be reused by a subsequent allocation. Because there is no benefit to marking an z/OS UNIX file “not-in-use”, rather than deallocating it, you should always deallocate z/OS UNIX files.

In addition, the system keeps track of data set use, and knows which data sets have not been in use for the longest time. These data sets might be deallocated automatically when the number of allocations for a step exceeds the control limit. See “Control limit” and “Permanently allocated attribute” on page 540 for more information on when and why a data set is deallocated automatically.

Removing the in-use attribute by dsname or ddname

Remove the in-use attribute by specifying verb code 02 with text unit key 8 in the SVC 99 parameter list. This function marks the data set as not-in-use. Marking a data set not-in-use makes it eligible for automatic deallocation if the control limit is exceeded.

If you code verb code 02 without specifying key 7 or key 8, dynamic unallocation **removes the in-use attribute** from data sets allocated through JCL, the TSO/E ALLOCATE command, or dynamically with the permanently allocated option; dynamic allocation *releases* data sets that were allocated dynamically without the permanently allocated option.

You use key 7 and key 8 to specify explicitly the type of processing you prefer. An explicit specification is satisfied in all but one case: dynamic allocation will not remove the in-use attribute from a non-permanently allocated, non-ampcdsname data set with a disposition of DELETE. Such a resource cannot be used to satisfy a subsequent request, so it is released.

Removing the in-use attribute by task ID

In addition to requesting removal of the in-use attribute by specifying a ddname or dsname, you may request, through verb code 05, that the in-use attribute be removed based on task ID. The attribute may be removed from all resources associated with a specified task, or all resources except those associated with the current task, its higher-level tasks, and the initiator.

Control limit

The **control limit** limits the number of data sets or z/OS UNIX files that can be allocated but marked “not-in-use.”

This control limit is determined by the JCL parameter DYNAMNBR on the EXEC statement and the number of DD statements. If the control limit is exceeded when an application program requests a new dynamic allocation, the system automatically attempts to deallocate enough data sets or z/OS UNIX files to meet the control limit, starting with eligible data sets or z/OS UNIX files that have been not-in-use for the longest time.

If the control limit is still exceeded after all eligible resources have been deallocated, the request for a new allocation fails. In this case, you must explicitly request deallocation of an existing allocation before the new allocation can be satisfied. Refer to the *z/OS MVS JCL User's Guide* for information on choosing a value for the DYNAMNBR parameter.

Note that if an allocation done with a DD statement is not OPEN, it is considered not in-use and is counted in the current number of not in-use allocations. If it is OPEN, it is **not** counted in the current number of not in-use allocations.

Permanently allocated attribute

The **permanently allocated attribute** prevents the system from automatically deallocating a particular data set or z/OS UNIX file to meet the control limit. The effect of this attribute is to determine eligibility for automatic deallocation.

The permanently allocated attribute is assigned automatically to data sets or z/OS UNIX files allocated through JCL and the TSO/E ALLOCATE command. In addition, you can request (through the DYNALLOC macro parameter list) that a data set be assigned this attribute when you dynamically allocate the data set.

Note: Because permanently allocated resources are not automatically deallocated, and all resources allocated through the TSO/E ALLOCATE command and JCL are permanently allocated, the control limit is the primary factor that controls the number of resources that a TSO/E user can have allocated at the same time. For information on setting a control limit, see “Control limit” on page 539.

Convertible attribute

Because a data set requested by an application program might be allocated already, dynamic allocation first checks for an existing allocation that matches the current request. This check avoids redundant allocation processing. In some cases, an existing allocation matches the current request except for some parameters. Dynamic allocation can change certain unmatching parameters of the existing allocation to meet the current request if the existing allocation has the **convertible attribute** and the data set is not in use. The convertible attribute allows dynamic allocation to change the following parameters of the existing allocation:

- Ddname
- Member name
- Status
- Normal disposition
- Conditional disposition
- Space
- Deallocation at CLOSE
- Input only
- Output only
- DCB attributes
- Password
- Permanently allocated attribute.

The convertible attribute is assigned automatically to all data sets that are dynamically allocated without the permanently allocated attribute. You can, however, assign both the convertible attribute and the permanently allocated attribute to a resource; although you might want to prevent a data set from being automatically deallocated, you might also want to allow some of its parameters to be changed to satisfy a new allocation.

Using ddname allocation

Ddname allocation allows you to reuse, by specifying only the associated ddname, a previously allocated data set that was marked not-in-use. Ddname allocation causes the system to assign the in-use attribute to the data set.

You request dynamic allocation by ddname by specifying verb code 06 and putting the ddname to be allocated in the DYNALLOC macro parameter list. For the system to satisfy your ddname dynamic allocation request, the existing allocation must not be in use. In addition, it must not have the convertible attribute; or it must be permanently concatenated. In other words, it must have properties that ensure that the ddname could not have been disassociated from the existing allocation. (See "Requesting the permanently concatenated attribute" on page 536 for a description of this attribute.)

If the existing allocation with the specified ddname does not meet these requirements, or if the ddname is not associated with any of your program's existing allocations, the system fails the request and returns an error reason code in the SVC 99 parameter list.

If the existing allocation meets the requirements, the system assigns it the in-use attribute and the request has been satisfied. If the existing allocation is a member of a concatenated group, all members of the group are assigned the in-use attribute, so the entire group has been allocated.

To request that the system return an indication of whether the existing allocation that satisfies the request is associated with a dummy data set, use verb code 6 with key 2.

Installation options for DYNALLOC macro functions

This information describes the types of default values and options your installation might want to use to control dynamic allocation processing. These values and options include:

- Default values for space, unit, and TIOT information
- Mounting volumes and bringing devices online
- Installation validation routine.

Using default values

For information about using the following default values for dynamic allocation processing, see the ALLOCxx parmlib member in *z/OS MVS Initialization and Tuning Reference*:

- Space values (SPACE keyword)
- Unit information (UNIT keyword)
- TIOT values (TIOT keyword).

Mounting volumes and bringing devices online

Dynamic allocation processing can bring devices online and have volumes mounted.

This function is optional for TSO/E users, because it is time-consuming and requires operator intervention; it is not always desirable in an interactive environment. If selected, the option is assigned through the UADS entries.

By default, other users of DYNALLOC can always have volumes mounted and devices brought online. The ALLOCxx member of SYS1.PARMLIB, described in *z/OS MVS Initialization and Tuning Reference*, defines the defaults for mounting volumes and bringing devices online. If you do not want volumes mounted or devices brought online during dynamic allocation, you can indicate it in the SVC 99 parameter list.

In addition, the operator may inform dynamic allocation that a volume is not to be mounted or that a device is not to be brought online. If the operator prevents the mounting or bringing online of a volume or device, the allocation request fails.

If you allow volume mounting, dynamic allocation waits for tape volumes to be mounted. Batch allocation processing, by contrast, does not wait for tape volumes to be mounted. When a volume is mounted for a dynamic allocation request, the system verifies that it is the correct volume when it opens the data set.

If the option to have volumes mounted and devices brought online is not in effect, tape and direct access devices that have an outstanding mount request, or that are not ready, are not eligible for use by dynamic allocation.

Installation input validation routine for dynamic allocation

An exit (IEFDB401) from dynamic allocation allows an installation-written routine to validate or alter any DYNALLOC request.

Through IEFDB401, your installation can:

- Control the amount of direct access space requested
- Check for authorization to use specified units
- Check for authorization to use specified data sets
- Check for authorization to hold certain resources for reuse.

See *z/OS MVS Installation Exits* for information about IEFDB401.

Chapter 26. Requesting dynamic allocation functions

This information explains how to code a dynamic allocation request.

To submit a dynamic allocation request, read the following topics:

- "Building the SVC 99 parameter list" describes what you must code to submit the dynamic allocation request, and what information the system returns to you.
- "SVC 99 parameter list verb codes and text units, by function" on page 625 describes how to code text units to supply the parameters for a dynamic allocation request. The text units are listed by function.

To diagnose errors in your dynamic allocation request, see "Processing messages and reason codes from dynamic allocation" on page 555, which describes the return and reason codes, and supplies actions for correcting the error.

To process dynamic allocation messages and reason codes, see "Processing messages and reason codes from dynamic allocation" on page 555, which describes how to set the fields of the request block extension to request message processing, and how to process the messages you receive from dynamic allocation.

If you are unfamiliar with dynamic allocation, or are not sure that dynamic allocation is right for your program or your programming environment, read Chapter 25, "Dynamic allocation," on page 521 before using this information to code a dynamic allocation request. This information describes allocation, when to use dynamic allocation and when not to, and which dynamic allocation functions are meaningful in a batch or an interactive environment.

Note:

BPXWDYN is a text interface to a subset of the SVC 99 (dynamic allocation) and SVC 109 (dynamic output) services. BPXWDYN supports data set allocation, unallocation, concatenation, and the addition and deletion of output descriptors. BPXWDYN is designed to be called from REXX, but it may be called from several other programming languages, including Assembler, C, and PL/I. For more information, see *z/OS Using REXX and z/OS UNIX System Services*.

Building the SVC 99 parameter list

You request dynamic allocation functions by coding the DYNALLOC macro. The DYNALLOC macro has no operands. To use it, you must supply information about your request to the system by using the SVC 99 parameter list. "Building the SVC 99 parameter list" describes the following steps for submitting your dynamic allocation request:

To determine the amount of storage that you'll need for your dynamic allocation request, see "Obtaining storage for the parameter list" on page 545.

To construct your parameter list, see "Mapping storage for the parameter list" on page 546, which describes the mapping macros you need to use to map the storage you've obtained.

To submit your dynamic allocation request, see the following information:

- “Setting up the request block pointer” on page 547, which describes setting up the pointer to the request block
- “Setting up the request block” on page 547, which describes how to set the fields in the request block for your request
- “Setting up the text unit pointer list” on page 554, which describes how to set the text unit pointer list to the addresses of your text units
- “Setting up the text units” on page 553 and “SVC 99 parameter list verb codes and text units, by function” on page 625 to code the text units that provide the specifics for your dynamic allocation request.

Coding a dynamic allocation request

The following table describes what you must code to submit a dynamic allocation request.

Code:	**To:**
IEFZB4D0 mapping macro	Construct the parameter list.
IEFZB4D2 mapping macro	Provides mnemonics for the text units.
Request block pointer	Specify the address of the request block.
Request block	Indicate which dynamic allocation function you want, how the system should fulfill your request, and point to other information about the request.
Request block extension	Use message processing and receive reason codes.
Text Unit Pointer List	Specify the addresses of the text units.
Text Units	Tell the system the specifics of the request.

The request block indicates the function you want DYNALLOC to perform, and the request block extension contains message processing information and fields into which the system returns reason codes. The text unit pointers are the addresses of the text units, and the text units contain the keys and parameters for the dynamic allocation functions. Figure 67 on page 545 illustrates the structure of the SVC 99 parameter list.

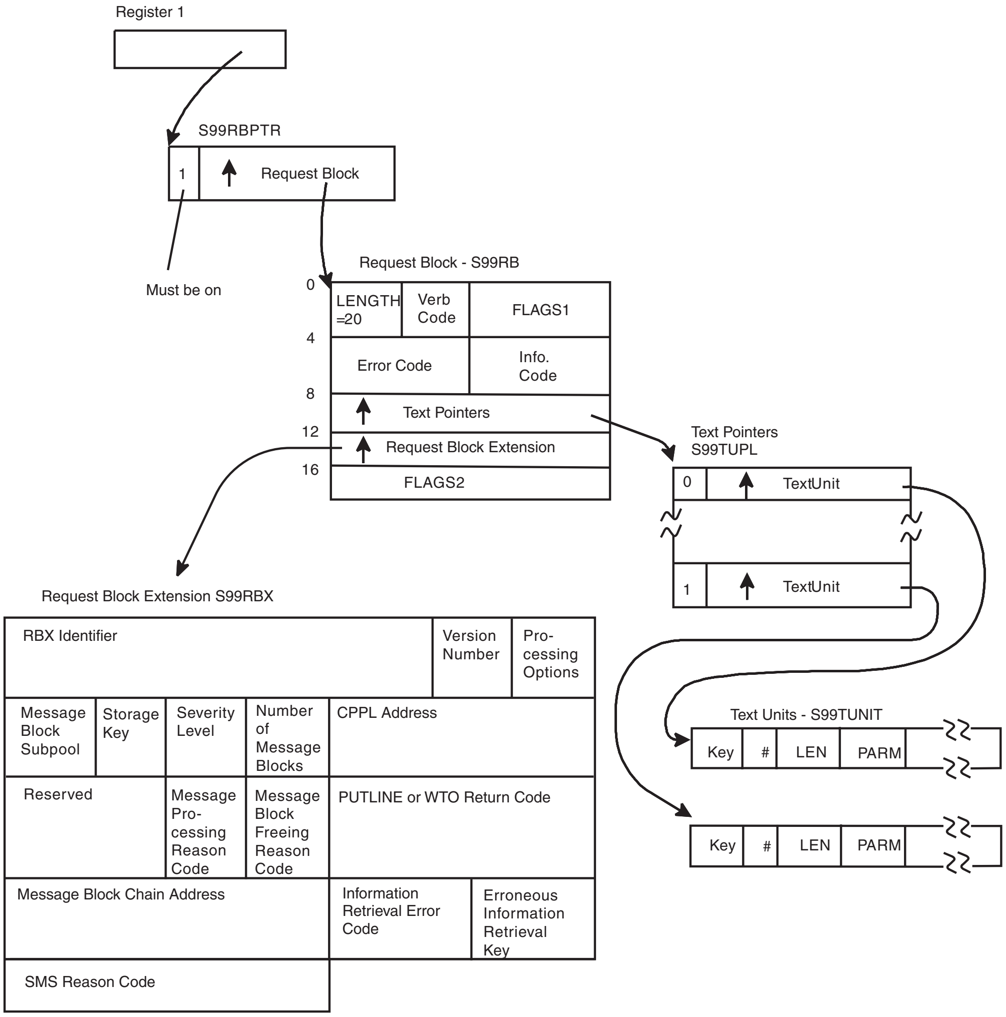


Figure 67. Structure of the SVC 99 Parameter List

Obtaining storage for the parameter list

The amount of storage that you need for your parameter list depends on the number and type of text units you need to describe your request to the system. At a minimum, you will need storage for the request block (S99RB), the request block extension (S99RBX), the text unit pointer list, and possibly, some of the text units.

The text units into which the system returns information must be in dynamic storage. These include information retrieval keys (described in “Dynamic information retrieval text units” on page 697) and some of the non-JCL dynamic allocation functions keys (described in “Non-JCL dynamic allocation functions” on page 678). Use the GETMAIN or STORAGE macros to obtain storage for these text units.

Text units that the system does **not** use to return information need not be in dynamic storage. They can be in static storage in your program.

Estimating the amount of storage for your request

The following guidelines will help you decide how much storage to request. Table 70 on page 546 outlines the minimum amount of storage required for a request that includes information retrieval text units. You might want to obtain enough storage for *all* of your text units; if so, increase the amount of storage you request to accommodate your additional text units.

Table 70. Minimum Amount of Storage for a Dynamic Allocation Request

Amount of storage (in bytes)	**Purpose**
20	Construct the RB (described in "Setting up the request block" on page 547) to submit your dynamic allocation request.
36	Construct the RBX (described in "Setting up the request block extension" on page 555) to use message processing and SJF information reason codes.
4 per text unit	Construct the text unit pointer list with the addresses of your text units. Each text unit pointer is 4 bytes long. Multiply 4 times the number of text units to determine the amount of storage you need for the text unit pointer list.
4 + (2 × *number of text unit PARM fields*) + (*length of all text unit PARM fields*) per information retrieval text unit	Construct your text unit (described in "SVC 99 parameter list verb codes and text units, by function" on page 625). For each information retrieval text unit, you will need 4 bytes to contain: • The text unit key • The number of length and parameter pairs. For each information retrieval text unit, you require another 2 bytes to contain the length (LEN) of the parameter (PARM) field. If you code multiple PARM fields, you need 2 bytes for *each* PARM field you code. In addition, you need enough storage for the text unit PARMs.

Characteristics of storage for parameter list

Storage for the parameter list must begin on a fullword boundary. The entire parameter list structure must be created in storage with the same key as the one for the caller of DYNALLOC. In addition, the request block, the extension, and any information retrieval text units must be in non-store-protected storage. This requirement prevents an 0C4 abend when DYNALLOC stores the information retrieval text units into the caller's storage.

Clearing the storage: It is a good programming practice to clear the storage you obtain before using it. The GETMAIN and STORAGE macros do not always clear the storage for you.

Passing the address of the obtained storage to DYNALLOC

On entry to DYNALLOC, register 1 must contain the address of a pointer to the request block. The STORAGE and GETMAIN macros return the address of the obtained storage in general purpose register 1.

Mapping storage for the parameter list

IBM supplies mapping macros to aid in constructing the SVC 99 parameter list. They are:

Mapping Macro	**Purpose**
IEFZB4D0	Provides symbolic names (DSECTs) for the positional information in the structure.
IEFZB4D2	Provides mnemonics for the text units.

The names in Figure 67 on page 545 are those assigned by the macro IEFZB4D0. For a complete list of the fields mapped by IEFZB4D2 and IEFZB4D0, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Note: DYNALLOC stores only the information retrieval text units originally specified by the caller and validated by the IEFDB401 exit routine (see *z/OS MVS Installation Exits*). Text units added or modified by IEFDB401 are not stored in the calling program's storage.

Setting up the request block pointer

The request block pointer S99RBPTR is a single fullword containing the address of the request block. The high-order bit in this field must be set to one.

Place the address of the storage you obtain for the request block into the request block pointer.

Setting up the request block

The request block consists of fields in which you provide information about your request, and into which the system stores information about the success of the request. The following table indicates the fields into which you must place a value and those that the system uses to return information.

RB Fields That You Must Initialize	RB Fields Into Which the System Returns Information
S99RBLN S99VERB S99FLAG1 S99TXTPP S99FLAG2 S99S99X	S99ERROR S99INFO

The request block is required input to DYNALLOC. It must begin on a fullword boundary. Mapping macro IEFZB4D0 assigns it a DSECT name of S99RB. It contains the following fields (the names in parentheses are those assigned by IEFZB4D0):

S99RB fields

Length (S99RBLN): A one-byte field containing the length of the request block. Determine the length of the request block by calculating the mathematical difference between the address of S99RBEND (see "S99RBEND" on page 552) and the address of S99RB.

Verb code (S99VERB): A one-byte field that identifies the dynamic allocation function to be performed. You must specify one of the following verb codes:

Verb Code	Name	Meaning
01	S99VRBAL	Request for dsname allocation
02	S99VRBUN	Request for deallocation (based on dsname or ddname)
03	S99VRBCC	Request for concatenation
04	S99VRBDC	Request for deconcatenation
05	S99VRBRI	Request for removing the in-use attribute based on task-ID
06	S99VRBDN	Request for ddname allocation
07	S99VRBIN	Request for information retrieval

Note: For a matrix version of S99NOMNT, S99OFFLN and S99MOUNT interrelationships see Table 74 on page 552 and Table 75 on page 553.

Flags (S99FLAG1): A two-byte field that instructs the system on how to satisfy dynamic allocation requests. S99FLAG1 bit settings (except for S99CNENQ and S99MSGL0) are used only for dsname allocation requests. The bits in the field are as follows:

Table 71. S99FLG11 — First byte of S99FLAG1

Bit	Bit Name	Meaning When On
0	S99ONCNV	Only use an existing allocation that has the convertible attribute to satisfy the request.
1	S99NOCNV	Do not use an existing allocation to satisfy this request.
2	S99NOMNT	Do not mount volumes or consider off-line devices. (This bit overrides S99MOUNT and S99OFFLN in S99FLAG2.) If this bit is on and the request causes a private catalog to be allocated, mounting will not be allowed for that catalog.
3	S99JBSYS	Used for SYSOUT data sets, this flag indicates that the system is to treat the data set as part of the job's normal output. The data set is not expected to be dynamically deallocated (spun off). If the data set is dynamically deallocated, it will be printed immediately, but paging space will not be released until the job ends.
4	S99CNENQ	Issue a conditional ENQ on the TIOT resource. If the TIOT is not available, an error code is returned to the user.
5	S99GDGNT	When the bit is on, the system is to use a LOCATE to determine the relative generation number based on the most recent catalog information. The relative generation number will reflect GDG data sets created or deleted by other jobs in a multitasking environment. When the bit is off, the system is to determine the relative generation number based on the catalog information that was available the first time the GDG was referenced during the job or TSO/E session. The relative generation number will reflect only those GDG data sets created or deleted by this job. See *z/OS MVS JCL User's Guide* for more information about GDG data sets, and Table 73 on page 549 for an example using S99GDGNT.
6	S99MSGL0	Requests that no messages be issued for this dynamic allocation. Use this bit together with S99ERMSG (described in bit 1 of “Processing options (S99EOPTS)” on page 555) to obtain necessary messages. This bit overrides the MSGLEVEL parameter on the JOB card of the JCL with MSGLEVEL=(,0). Specifying S99EIMSG (described in bit 0 of “Processing options (S99EOPTS)” on page 555) for a dynamic allocation will override the specification of S99MSGL0.
7	S99NOMIG	Do not recall migrated data sets, referenced either directly (for example, through text unit DALDSNAM) or indirectly (for example, through text units DALDCBDS or DALVLRDS). When the volume on which the data set resides is migrated and S99NOMIG is on, the CATALOG facility returns the volume name MIGRAT to allocation, and the system fails the allocation request with error reason code x'278'. If you are using a product other than DFHSM, ensure that it returns the volume name MIGRAT to the CATALOG facility under these conditions.

Table 72. S99FLG12 — Second byte of S99FLAG1

Bit	Bit Name	Meaning When On
8	S99NOSYM	Disable symbolic substitution for the current request.
9	S99ACUCB	Use 4 byte actual UCB addresses, which is known as the NOCAPTURE option. **Note:** 1. VSAM, BSAM, BPAM, QSAM and EXCP support the S99ACUCB option. 2. The S99ACUCB “NOCAPTURE” option causes creation of an XTIOT even if S99TIOEX has not been coded. S99ACUCB does not require authorization as S99TIOEX does. 3. When set, S99DXACU overrides this flag.
10	S99DSABA	Request that the DSAB for this allocation be placed above the 16MB line. **Note:** 1. S99TIOEX (authorized users only) must be set to get an above-the-line DSAB. 2. VSAM, BSAM, BPAM, QSAM and EXCP support the S99DSABA option. 3. When you use S99DSABA you need to understand how the DSAB that represents the DD allocation statement is queued. Most programs use the IEFDDSRV or GETDSAB interfaces with LOC=ANY to locate information about the DD allocation; however, programs before z/OS V1R13 might access the DSAB without using these services. When you set S99DSABA, you must update your code either to use the IEFDDSRV or GETDSAB interfaces or to use the proper queuing fields in the DSAB mapping. 4. When set, S99DXACU overrides this flag.
11	S99DXACU	Request that the DSAB for this allocation be placed above the 16MB line, that the 4 byte actual UCB address, which is known as the NOCAPTURE option, be used and that an XTIOT be created. This results in bypassing the TIOT limits and below-the-line storage constraints associated with concurrently allocating many data sets. When this bit is set, the settings of S99DSABA, S99TIOEX and S99ACUCB are ignored. **Note:** To retrieve information about the allocation, you must specify LOC=ANY on the GETDSAB or IEFDDSRV services. These services are recommended instead of coding TIOT or DSAB search routines, as queuing is different for above the line DSABs and TIOTs.
12 - 15		Not an intended programming interface; set to zero.

Table 73. Example of Using the S99GDGNT bit of the S99FLAG1 field

Job 1		**Job 2**
DYNALLOC allocates GDG.XX(+1) as DD1	Event A	
DYNALLOC unallocates DD1	Event B	
	Event C	Allocate GDG.XX(+1), either through JCL or DYNALLOC

Table 73. Example of Using the S99GDGNT bit of the S99FLAG1 field (continued)

Job 1		Job 2
	Event D	Job ends
DYNALLOC allocates GDG.XX(+1)	Event E	
Events occur in alphabetical order. At the beginning of Job 1, the base GDG is GDG.XX.G0000V00:		
With S99GDGNT flag off for Event E: • Event A allocates GDG.XX.G0001V00. • Event C allocates GDG.XX.G0002V00. • Event E allocates GDG.XX.G0001V00.		With S99GDGNT flag on for Event E: • Event A allocates GDG.XX.G0001V00. • Event C allocates GDG.XX.G0002V00. • Event E allocates GDG.XX.G0003V00.

Error code (S99ERROR): A two-byte field into which the system returns error reason codes when DYNALLOC has completed processing. See “Interpreting DYNALLOC return codes” on page 586.

Info code (S99INFO): A two-byte field that contains information reason codes when DYNALLOC has completed processing. See “Interpreting information reason codes from DYNALLOC” on page 587.

Text unit pointers' address (S99TXTPP): A fullword field that contains the address of a list of pointers to the text units. Initialize this field by storing into it the address of the list of pointers to the text units.

Request block extension address (S99S99X): A fullword field containing the address of the request block extension. If you want to use the message processing function and information reason codes provided by dynamic allocation, initialize this field with the address of the request block extension (RBX). If you don't want this capability, this field should contain zeroes. The RBX is described in “Processing messages and reason codes from dynamic allocation” on page 555.

Additional flags (S99FLAG2): A four-byte field of indicators. These indicators may be set only by authorized programs. To be authorized, the requesting program must meet at least one of the following criteria:

- It must have a PSW key of 0 - 7.
- It must be in supervisor state.
- It must be APF-authorized.

The S99FLAG2 indicators (except S99TIONQ) are used only for dsname allocation requests.

The S99FLAG2 bits are:

Bit	Bit Name	Meaning When On
0	S99WTVOL	Wait for volumes.
1	S99WTDSN	Wait for dsname. **Note:** In a JES3 environment, authorizing a dynamic allocation request to wait for data set availability might cause a system interlock.

Bit	Bit Name	Meaning When On
2	S99NORES	Do not reserve (serialize) data sets. **Note:** Data sets being allocated are normally serialized via ENQ with MAJOR name SYSDSN, MINOR name -data set name-. When S99NORES is set, there is NO data set serialization and multiple tasks may reference or update the data set simultaneously, resulting in unpredictable effects. It is the responsibility of the authorized program setting S99NORES to provide the necessary serialization.
3	S99WTUNT	Wait for units.
4	S99OFFLN	Consider off-line devices. The system ignores this bit if S99NOMNT in S99FLAG1 is on. For a matrix version of S99NOMNT, S99OFFLN and S99MOUNT interrelationships see Table 74 on page 552 and Table 75 on page 553.
5	S99TIONQ	Exclusive TIOT ENQ already performed in this ASID under this TCB with the SYSTEM option.
6	S99CATLG	Set special catalog data set indicators.
7	S99MOUNT	Volumes may be mounted. The system ignores this bit if S99NOMNT in S99FLAG1 is on. For a matrix version of S99NOMNT, S99OFFLN and S99MOUNT interrelationships see Table 74 on page 552 and Table 75 on page 553.
8	S99UDEVT	Unitname parameter for DALUNIT is a device type. If you are using the output from the DEVTYPE macro, be sure the shared DASD bits are turned off.
9		Not an intended programming interface. Must be zero.
10	S99DYNDI	Do not perform JES3 data set integrity processing.
11	S99TIOEX	Build XTIOT entry. **Note:** 1. The XTIOT is a non-contiguous TIOT entry that is not accessible through the contiguous TIOT. 2. VSAM, BSAM, BPAM, QSAM and EXCP support the S99TIOEX option. 3. Setting this bit requires APF authorization, supervisor state, or system key. 4. When set, S99DXACU overrides this flag.

Bit	Bit Name	Meaning When On
12	S99DASUP	Used by authorized programs to suppress the DD-level accounting. Setting this bit can affect the SMF data created for the following: • The EXCP section of SMF Record Type 30. • SMF Record Type 40. • SMF Record Type 14 for the fields SMF14NTR and SMF14NER. This bit is only recommended for programs allocating VSAM data sets with generated DD names, or when the exploiting program has established that the usefulness of the SMF data is less than the benefit to system performance. Because the data is used by an installation and suppressed by the exploiting program, an external switch controlling the program's use of this bit is strongly recommended.
13 - 31		Not an intended programming interface. Must be zero.

In using any of the following flags, the caller sets up a condition in which a job might wait for a resource to be available:

- S99WTVOL
- S99WTDSN
- S99WTUNT
- S99OFFLN
- S99MOUNT

Use care when you set these flags; setting any one of them might cause a deadlock situation. For example, consider the situation where JOBA owns a resource that JOBB wants and JOBB owns a resource that JOBA wants. If one of the above flags are on, the two jobs will wait until one job is cancelled. To avoid deadlocks, you might use an STIMERM macro across the call.

S99RBEND: A marker indicating the end of the request block.

Relationships among selected fields

Table 74. Relationship of S99NOMNT, S99OFFLN and S99MOUNT. (NON-TSO Dynamic)

S99NOMNT	S99OFFLN	S99MOUNT		ALLOW OFFLINES	ALLOW MOUNTS
0	0	0		Y	Y
0	0	1		N	Y
0	1	0		Y	N
0	1	1		Y	Y
1	any	any		N	N

Table 75. Relationship of S99NOMNT, S99OFFLN and S99MOUNT. (TSO User)

S99NOMNT	**UADS MNT OPT**	**S99OFFLN**	**S99MOUNT**		**ALLOW OFFLINES**	**ALLOW MOUNTS**
0	0	0	0		N	N
0	0	0	1		N	Y
0	0	1	0		Y	N
0	0	1	1		Y	Y
0	1	any	any		Y	Y
1	any	any	any		N	N

Setting up the text units

The text unit consists of fields in which you provide information about your request. The system uses text unit fields to return information only when you request an information retrieval function.

Each text unit is a variable-length field (assigned the DSECT name S99TUNIT by macro IEFZB4D0) that contains the following subfields:

Key (S99TUKEY)

A two-byte field in which you specify a hexadecimal number that identifies the function that the DYNALLOC macro is to perform. DYNALLOC ignores a KEY field of zero.

Number (S99TUNUM): A two-byte field in which you specify, in hexadecimal, the number of length and parameter combinations in the text unit.

Combination (S99TUENT): The label for length and parameter combinations. IEFZB4D0 provides a separate DSECT (named S99TUFLD) for use when specifying multiple parameters in a single text unit. This DSECT places the length field at displacement 0 for the second and subsequent combinations:

S99TUFLD
Label for the DSECT

S99TULEN
Label for the length field

S99TUPRM
Label for the parameter

Length (S99TULNG): A two-byte hexadecimal number specifying the length of the parameter field (S99TUPAR) that follows. For information retrieval text units, the system uses this field to indicate the length of the information it is returning, or to indicate that the information was not available, by setting it to 0.

Parm (S99TUPAR): A variable-length field in which you put the parameter information identified by the value in the KEY field. See “SVC 99 parameter list verb codes and text units, by function” on page 625 for a description of the values you can code for each text unit key.

Structure of the text units

When setting up your text units, follow the rules below. You will find rules for coding specific text units in “SVC 99 parameter list verb codes and text units, by function” on page 625.

Using special characters in text units: In general, special characters, such as those requiring apostrophes in JCL statements, are not valid in PARM values. However, you may code special characters:

- In the DALUSRID text unit
- In a dsname, when specifying the dsname in apostrophes.

Using trailing blanks in text units: Parameters whose values consist of alphameric and national (@, $, #) characters may include trailing blanks.

Order of text units: The text units may be in any order.

Independence of text unit key meaning across DYNALLOC functions: Each dynamic allocation function has an associated set of text unit keys, and each set is independent of any other. For example, the functions of allocation and unallocation may both use a KEY value of X'0007', but that value does not necessarily have the same meaning for both functions.

Setting up the text unit pointer list

The text unit pointer list is a variable-length list of fullwords containing pointers to the text units. Mapping macro IEFZB4D0 assigns the DSECT name S99TUPL to the list, and the label S99TUPTR to each pointer in the list.

Establishing addressability to the text unit pointer list

One way to establish addressability to the text unit pointer list is to find the address at the end of the request block (by adding S99RB and S99RBLEN) and base S99TUPL at this address.

You might want to set up your SVC 99 parameter list differently. For example, you might want the request block extension to follow the request block, instead of the text unit pointer list. You can do so as long as your pointers are correct.

Filling in the text unit pointer list

Once you have established addressability to the text unit pointer list, you can fill in the addresses of the text units. Do this by storing the address of the first text unit into the first word of the text unit pointer list, moving to the next text unit pointer in the list, and storing the address of the second text unit in it. Continue until you have stored the addresses of all of your text units in the list; then indicate the end of the text unit pointer list.

The system ignores a fullword of zeros in the text unit pointer list.

Setting the end of list indicator

You indicate the end of the list by setting the high-order bit of the last pointer to one. Mapping macro IEFZB4D0 assigns label S99TUPLN to an equate that allows you to turn on the end-of-list indicator.

Failure to set the end of list indicator can cause dynamic allocation to interpret data as addresses and cause a storage exception.

Processing messages and reason codes from dynamic allocation

To help you diagnose errors that can occur when using dynamic allocation functions, dynamic allocation supplies information reason codes and IBM supplies an extended message processing program. You must code the request block extension (RBX) to receive reason codes in addition to those returned in the S99ERROR and S99INFO request block fields, and to use the message processing function.

Setting up the request block extension

The RBX consists of fields in which you provide information about your request, and into which the system stores information about the success of the request. The following table indicates the fields into which you must place a value and those that the system uses to return information.

RBX Fields That You Must Initialize	RBX Fields Into Which the System Returns Information
S99EID S99EVER S99EOPTS S99ESUBP S99EKEY S99EMGSV S99ECPPL	S99ENMSG S99ERCO S99ERCF S99EWRC S99EMSGP S99EERR S99EINFO S99ERSN

The request block extension must begin on a fullword boundary. Mapping macro IEFZB4D0 assigns it a DSECT name of S99RBX. It contains the following fields (the names in parentheses are those assigned by IEFZB4D0):

S99RBX fields

Request block extension identifier (S99EID): A six-byte field containing the request block extension identifier. In your program, define the string 'S99RBX' as a character constant with a length of 6, and move in into the S99EID field.

Version (S99EVER): A one-byte version number of the request block extension. Set S99EVER to the constant S99RBXVR in IEFZB4D0.

Processing options (S99EOPTS): A one-byte field that defines the dynamic allocation message processing options. Select the processing options that you want by setting bits in this field as follows:

Bit	Bit Name	Meaning When On
0	S99EIMSG	The system issues error messages **before** control returns to the caller of DYNALLOC. This bit **must** be set on if Bit 5 (S99EWTP) is set on.
1	S99ERMSG	The system returns the unformatted messages in message blocks to the caller of DYNALLOC. The system does not issue the messages unless S99EIMSG is set.
2	S99ELSTO	The system returns the message blocks to the caller in virtual storage below 16 megabytes.

Bit	Bit Name	Meaning When On
3	S99EMKEY	The caller has specified a storage key in S99EKEY. When building message blocks, the system builds them in a storage area whose key is equal to the key specified in S99EKEY.
4	S99EMSUB	The caller has specified a storage subpool in S99ESUBP. The system builds the message blocks in that subpool when S99EMSUB is set.
5	S99EWTP	If this bit is set, Bit 0 (S99EIMSG) must also be set, and the system uses a WTO macro to issue the error messages. Otherwise, the system uses a TSO PUTLINE command to issue the messages.
6 - 7		Not an intended programming interface. These bits must be zero.

Note:

1. If your program requested that message blocks be returned from the SVC 99 request, the storage that contains those blocks has a storage key that matches the PSW key value of your program, unless you overrode the storage key using the S99EKEY or S99ESUBP field.
2. If message blocks are returned, your calling program must ensure that the message blocks are deleted (via FREEMAIN). Failure to do so will cause the message blocks to accumulate in storage, eventually leading to an out-of-storage condition. You may use the IEFDB476 program to process the message blocks. (See “Using the functions of the IEFDB476 program” on page 580). Even if you choose not to use IEFDB476 to manage the process, you may still invoke the program simply to free the message blocks.
3. Programs attempting to process message blocks using the IEFDB476 program must run with a PSW key value of zero, or with the key that matches the storage key of the storage that contains the message blocks.

Message block subpool (S99ESUBP): A one-byte field in which you specify the subpool containing the message blocks returned to the caller. The system ignores this field unless you have indicated, by setting bit S99EMSUB in the field S99EOPTS, that you will be specifying a storage subpool. If you do not indicate a subpool, the system uses a default subpool of 0. The valid subpools are subpools 0-255, and they must be specified in binary. An unauthorized program cannot request a system subpool.

Storage key (S99EKEY): A one-byte in which you specify the storage key for the storage in which the message blocks are returned. The system ignores this field unless you have indicated, by setting bit S99EMKEY in the field S99EOPTS, that you will be specifying a storage key. If you do not supply a storage key, the system uses the same key as the caller's TCB. The valid keys are 0-15, and they must be specified in binary. An unauthorized program cannot request a storage key that is different from its own key.

Severity level (S99EMGSV): A one-byte field in which you indicate the minimum severity of the messages that should be processed by DYNALLOC. The severity levels, which are ***informational***, ***warning***, and ***severe***, are defined by S99XINFO, S99XWARN, and S99XSEVE in the IEFZB4D0 mapping macro.

Number of message blocks returned (S99ENMSG): A one-byte field in which the system indicates the number of message blocks returned from DYNALLOC.

CPPL address (S99ECPPL): A fullword that contains the address of the command processor parameter list. This field is required if PUTLINE is used to issue messages. See *z/OS TSO/E Programming Guide* for information about the CPPL.

Not an intended programming interface (S99ERCR): A one-byte reserved field containing zeroes.

Not an intended programming interface (S99ERCM): A one-byte reserved field containing zeroes.

Message processing reason code (S99ERCO): A one-byte reason code by which the system indicates the failure of a message processing function.

Hexadecimal Code
Meaning and Action

03 **Meaning:** WTO failed.

Application Programmer Action: S99ERWC contains the return code from WTO. See *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for the meaning and action for the specific return code.

04 **Meaning:** PUTLINE failed.

Application Programmer Action: S99ERWC contains the return code from PUTLINE. See *z/OS TSO/E Programming Services* for the meaning and action for the specific return code.

05 **Meaning:** Unable to obtain storage for message blocks.

Application Programmer Action: Resubmit the request. If the problem recurs, consult the system programmer to determine why storage is unavailable.

06 **Meaning:** Unable to obtain storage for PUTLINE macro.

Application Programmer Action: Resubmit the dynamic allocation or IEFDB476 request. If the problem recurs, consult the system programmer to determine why storage is unavailable.

07 **Meaning:** A CPPL address was not supplied for the PUTLINE message output function. By invoking one of the following, you have requested that the system issue messages through PUTLINE.

- DYNALLOC with the S99EIMSG flag on and the S99EWTP flag off
- IEFDB476 with EMPUTLIN flag on and a value of 50 in EMIDNUM.

Application Programmer Action: See *z/OS TSO/E Programming Guide* for information about the CPPL.

08 **Meaning:** The message block chain was not valid. This is a system error, possibly a storage overlay.

Application Programmer Action: Ask your system programmer to contact the appropriate IBM support personnel.

09 **Meaning:** Message extraction failed because the message block chain was not valid.

Application Programmer Action: Verify that the message blocks to which S99EMSGP points have not been modified by your code. If they have not, this might be a storage overlay error. Ask your system programmer to contact the appropriate IBM support personnel.

0A **Meaning:**Message extraction was requested through bit EMRETURN in the EMFUNCT field, but no message buffer address was supplied.

Application Programmer Action: Supply the message buffer address in field EMBUFP and resubmit the request.

0B **Meaning:** The Dynamic Allocation Request was unsuccessful but no messages were returned. The reason is unknown.

Application Programmer Action: The S99RB field S99ERROR contains the Dynamic Allocation Error Reason Code. Using that reason code, locate the meaning and suggested actions in "Interpreting error reason codes from DYNALLOC" on page 591.

0C **Meaning:** Bit S99ERMSG in the RBX was not set on to request that the system return messages in the message block

Application Programmer Action: Set bit S99ERMSG in the RBX, or change the application program so that it does not invoke the IEFDB476 program when messages are not being returned.

0D **Meaning:** The Dynamic Allocation Request was successful, and no messages were returned. Change the application program so that it does not invoke the IEFDB476 program when no messages are expected.

Application Programmer Action: Change the application program so that it does not invoke the IEFDB476 program when no messages are expected.

Message block freeing reason code (S99ERCF): A one-byte reason code that explains why the system cannot free the message block storage area.

Hexadecimal Code
Meaning and Action

01 **Meaning:** A message block chain is not valid.

Application Programmer Action: Verify that the message blocks to which S99EMSGP points have not been modified by your code. If they have not, this might be a storage overlay error. Ask your system programmer to contact the appropriate IBM support personnel.

02 **Meaning:** A FREEMAIN failed.

Application Programmer Action: See *z/OS MVS Programming: Assembler Services Reference ABE-HSP* for reasons and actions for a FREEMAIN failure.

03 **Meaning:** Either the caller specified a storage key that was greater than 15, or an unauthorized caller specified a storage key that was different from the caller's key.

Application Programmer Action: Specify a valid key and resubmit the request. See Chapter 10, "Virtual storage management," on page 213 for information about storage keys.

04 **Meaning:** An unauthorized caller specified a system subpool.

Application Programmer Action: Specify a valid subpool and resubmit the request. See Chapter 10, "Virtual storage management," on page 213 for information about subpools.

PUTLINE/WTO macro return code (S99EWRC): The fullword return code from the WTO macro or PUTLINE macro.

Message block chain address (S99EMSGP):

NOT Programming Interface information

A full word that contains the address of a chain of message blocks.

End of NOT Programming Interface information

See "Processing messages from dynamic allocation" on page 580 for information about processing error messages.

Information retrieval error code (S99EERR): A two-byte code that the system returns to explain errors found in information retrieval text units of dynamic allocation. This two-byte code applies only to verb code 01, described in "Non-JCL dynamic allocation functions" on page 678. See "Interpreting DYNALLOC return codes" on page 586 for an explanation of the codes.

If there was a text unit in error, the field S99EINFO (described in "Information retrieval information code (S99EINFO)") will contain the text unit key.

The S99EERR code does not indicate whether the allocation was successful. The system indicates allocation errors in the request block fields S99ERROR and S99INFO. These fields are described in "Error code (S99ERROR)" on page 550 and "Info code (S99INFO)" on page 550.

Information retrieval information code (S99EINFO): A two-byte field in which the system returns an erroneous text unit key. The text unit is an information retrieval text unit. This two-byte code applies only to verb code 01.

SMS reason code (S99ERSN): A fullword field that contains the SMS reason code explaining the failure. The system returns a reason code in this field only when an error code beginning with X'97xx' is returned in the S99ERROR field of the request block.

IGD messages accompany the SMS reason code depending on the allocation message processing options in the S99EOPTS field of the request block. (See "Processing options (S99EOPTS)" on page 555.) The SMS reason code also corresponds to an IGD system message that further describes the error. For example, if the reason code is X'4379' (decimal 17273), the IGD message that contains the decimal equivalent, IGD17273I, describes the error.

If the IGD messages are suppressed, the following procedure could be followed to get further details:

- For error code X'970C', SMS creates entries in the logrec data set. Because there are no problem IDs associated with SMS reason codes, locate the logrec data set entry for the error that corresponds to the reason code.
- For error codes other than X'970C', SMS will not always create entries in the logrec data set. Try to recreate the error by executing in batch mode instead of using dynamic allocation.

The system returns a reason code in S99ERSN when a failure occurs during SMS processing. For any reason code returned in S99ERSN that is not described in the following list and that does not convert to a recognizable IGD system message, search the decimal code under Storage Management Subsystem reason codes in *z/OS DFSMSdfp Diagnosis*. If you still cannot locate the reason code, have the system programmer locate the logrec data set entry associated with the error and report the failure to the IBM support center.

Hexadecimal Code (Decimal)
Meaning and Action

12C (300)

Meaning: An abend occurred during storage management subsystem processing. Message IGD300I will be issued to the operator with a symptom dump.

System Programmer Action: Examine message IGD300I and the symptom dump to determine why the abend occurred.

12D (301)

Meaning: The data set is not eligible for allocation on an SMS volume.

Application Programmer Action: Determine whether the request is attempting to allocate to an SMS volume a data set that is not managed by SMS. Do not attempt to allocate the following types of data sets to an SMS volume:

- ISAM data set
- SYSOUT data set
- Subsystem data sets
- TSO/E data sets coming from or going to a terminal
- In-stream data sets
- Data sets having the same name as a previously-cataloged data set
- Data sets that are not cataloged in the integrated catalog facility (ICF) catalog

12E (302)

Meaning: The selected storage class requires that explicitly specified volumes are to be honored. The requirement to honor explicit volumes cannot be met for one of the following reasons:

- Not all of the volumes are SMS-managed
- Not all of the volumes are defined to the same storage group
- The storage group containing the volumes was not selected for this data set allocation.

Application Programmer Action: If specific volumes are not required, remove the explicit volume specification and reissue the request. If the explicitly specified volumes are required, contact the system programmer for assistance.

System Programmer Action: Make sure that all of the volumes are SMS-managed and are defined to the same storage group. Also, make sure the volumes have the properties that this storage group requires; or modify the storage class routine to select a non-guaranteed space storage class. Then have the application programmer reissue the request.

12F (303)

Meaning: A storage class was not derived for the data set. Therefore, the data set is not SMS-managed. The system ignores the specified management class.

Application Programmer Action: If the data set is supposed to be SMS-managed, get help from the system programmer to determine why a storage class was not derived for the data set.

132 (306)

Meaning: An unexpected error occurred during storage management subsystem processing.

System Programmer Action: Use the logrec data set to obtain information about the error that occurred. Refer to *z/OS DFSMSdfp Diagnosis* for an explanation of the reason code and return code.

133 (307)

Meaning: An installation exit either:

- Ended abnormally with an abend code
- Returned an unknown return code

System Programmer Action: Use the logrec data set and SYS1.DUMPnn to determine why the installation exit failed.

134 (308)

Meaning: The storage class or management class was derived for a data set, but the owner is not authorized to use the specified storage or management class.

Application Programmer Action: Either obtain authorization to use the storage class or the management class or use a storage or management class that you are already authorized to use. Reissue the request.

135 (309)

Meaning: An SMS data set is not allowed in the scope of a JOBCAT or STEPCAT.

Application Programmer Action: One of two actions can be taken:

- If the storage class was explicitly specified, remove the JOBCAT, STEPCAT, or storage class specification to ensure that the data set is not SMS-managed. Then reissue the request.
- If the JOBCAT or STEPCAT is required and the storage class was installation derived, use a data set specification that will not create an SMS-managed data set when reissuing the request.

136 (310)

Meaning: MGMTCLAS or STORCLAS cannot be specified for a data set that is not eligible to be SMS-managed.

Application Programmer Action: Do not attempt to specify MGMTCLAS or STORCLAS for the following types of data sets, which are not managed by SMS:

- ISAM data set
- SYSOUT data set
- Subsystem data sets
- TSO/E data sets coming from or going to a terminal
- In-stream data sets
- Data sets having the same name as a previously-cataloged data set
- Data sets that are not cataloged in the integrated catalog facility (ICF) catalog

139 (313)

Meaning: DSNTYPE cannot be specified for a data set that is not eligible to be SMS-managed.

Application Programmer Action: Do not attempt to specify DSNTYPE for the following types of data sets, which are not managed by SMS:

- ISAM data set
- SYSOUT data set
- Subsystem data sets
- TSO/E data sets coming from or going to a terminal
- In-stream data sets
- Data sets having the same name as a previously-cataloged data set
- Data sets that are not cataloged in the integrated catalog facility (ICF) catalog

13A (314)

Meaning: The data class that was derived for the data set contains a DSNTYPE attribute that is not supported with the current level of the operating system.

System Programmer Action: Make sure the ACS routines will derive a data class with a DSNTYPE attribute that is supported with the current level of operating system.

13B (315)

Meaning: An attempt was made to read from an SMS-managed scratch tape volume. This is not allowed.

Application Programmer Action: If a scratch volume is derived from an SMS tape library, specify a volume of SCRTCH. If the tape volume is private, either eject the volume from the SMS tape library and access it outside the library, or contact the system programmer so that the status of the volume can be changed from scratch to private using IDCAMS or ISMF, and access it within the library.

13D (317)

Meaning: A DSNTYPE of PIPE was specified when the PATH keyword was not specified. This is not allowed.

Application Programmer Action: Add the PATH keyword, or remove the DSNTYPE of PIPE.

140 (320)

Meaning: An OPEN/MVS request cannot be processed because OPEN/MVS is not installed.

Application Programmer Action: Process OPEN/MVS requests only on systems that have OPEN/MVS installed.

FA0 (4000)

Meaning: The data set is processed as a non-SMS-managed data set because the volumes are not SMS-managed.

Application Programmer Action: No action is required.

FA1 (4001)

Meaning: An unexpected error occurred in catalog processing while attempting to locate a data set.

System Programmer Action: Locate the logrec data set entry for this error and determine the return code and the reason code for the catalog locate error under message IDC3009I.

FA2 (4002)

Meaning: The data set is an SMS-managed data set that is referenced in one of the following:

- A job containing a JOBCAT

- A step containing a STEPCAT
- Both a job containing a JOBCAT and a step containing a STEPCAT.

Application Programmer Action: Do one of the following:

- If there is no need for the JOBCAT or STEPCAT DD statements, remove them and resubmit the job.
- If a JOBCAT DD statement is specified, remove it and insert STEPCAT DD statements only on the required steps. Then resubmit the job.
- If a STEPCAT DD statement is necessary, then divide the step into several steps so the steps that do reference SMS-managed data sets do not contain a STEPCAT DD statement. Then resubmit the job.
- If none of the above apply, contact the system programmer.

System Programmer Action: Make sure all referenced ICF catalogs are connected to the system master catalog.

FA4 (4004)

Meaning: The catalog entry indicates that a data set is not SMS-managed, but an SMS-managed volume serial was specified.

Application Programmer Action: Do one of the following:

- Remove the VOL=SER, and reissue the request
- Specify the correct VOL=SER, and reissue the request

FA6 (4006)

Meaning: The system attempted to allocate a data set on two tape volumes which are in different system-managed tape libraries. Both volumes must be in the same library for the job to run. If more than two volumes are involved, an error is returned for the first mismatch encountered.

Application Programmer Action: If the volume serial numbers are specified incorrectly, then correct the specification. If the numbers are correct, then contact the tape librarian to ensure that all the tape volumes reside in the same library. When one of the two actions has been taken, reissue the request.

FA7 (4007)

Meaning: The system attempted to allocate a data set on a DASD volume and on a tape volume that resides in a system-managed tape library. Mixed media types are not allowed for a single data set. If more than two volumes are involved, an error is returned for the first incorrect allocation.

Application Programmer Action: If the volume serial numbers are specified incorrectly, then correct the specification. If the numbers are correct, then contact the tape librarian to eject the tape volumes from the system-managed tape library and access them on non-system managed tape drives.

FD3 (4051)

Meaning: The data set is processed as a non-SMS-managed data set because it was determined that the data set was an uncataloged special system data set.

Application Programmer Action: No action is required.

FD6 (4054)

Meaning: The specified data set was not found.

Application Programmer Action: Ensure that the data set name was specified correctly, and that the data set is cataloged in the appropriate catalog. Contact the system programmer to check if the alias entries point to the catalog. Reissue the request.

1324 (4900)

Meaning: An attempt to get the file status for a file in a z/OS UNIX file system failed.

Application Programmer Action: Submit a batch job to locate the z/OS UNIX file to determine the return and reason codes returned in the corresponding IGD04900I message.

1325 (4901)

Meaning: A path name was not specified for an allocation of a file in a z/OS UNIX file system during an SMS catalog services request.

Application Programmer Action: Add a path name to the requested allocation.

138C (5004)

Meaning: All volumes for a guaranteed space request are defined to SMS, but do not belong to the same storage group.

Application Programmer Action: Specify volumes that are within a single storage group, or remove volume specification and allow SMS to select volumes. Reissue the request.

1392 (5010)

Meaning: None of the volumes for a guaranteed space request are defined to SMS.

Application Programmer Action: Specify volumes that are defined to SMS and are within a single storage group, or remove volume specification and allow SMS to select volumes. Reissue the request.

1393 (5011)

Meaning: Some of the volumes for a guaranteed space request are not defined to SMS.

Application Programmer Action: Specify volumes that are defined to SMS and within a single storage group, or remove volume specification and allow SMS to select volumes. Reissue the request.

1B59 (7001)

Meaning: An unexpected error occurred in the catalog while attempting to roll a generation data set (GDS) into a generation data group (GDG).

System Programmer Action: Submit a batch job to roll the generation data set into the generation data group, and use the return and reason codes from any corresponding IGD messages when contacting the IBM support center to determine the cause of the catalog failure.

1B5A (7002)

Meaning: The storage management subsystem was invoked to delete a data set. The volume associated with that data set was either not defined to the configuration or was not currently mounted. The volume may have been deleted from the active configuration.

Application Programmer Action: Contact the system programmer to make the necessary modifications. Then delete the data set after the modifications have been completed.

System Programmer Action: Either modify the configuration to include the volume, or make sure the volume is online.

1EDC (7900)

Meaning: A path name is required for the allocation of a file in a z/OS UNIX file system during an SMS disposition processing request.

Application Programmer Action: Add a path name for the requested allocation.

1EDD (7901)

Meaning: An attempt to delete a z/OS UNIX file failed.

Application Programmer Action: Submit a batch job to delete the z/OS UNIX file to determine the return and reason codes returned in the corresponding IGD07901I message.

1EDE (7902)

Meaning: An invalid disposition was specified for a z/OS UNIX file.

Application Programmer Action: Correct the disposition for the z/OS UNIX file.

2B5C (11100)

Meaning: A dynamic allocation request specified the RECFM keyword with the KS, ES, RR, or LS value on the RECORG keyword. This combination is not allowed because the RECFM keyword only applies to non-VSAM data sets, while the KS, ES, RR, and LS values of the RECORG keyword only apply to VSAM data sets.

Application Programmer Action: Change or remove the RECORG value or RECFM keyword on the dynamic allocation request; then reissue the request.

2B5D (11101)

Meaning: A dynamic allocation request specified the DSNTYPE keyword with the ES, RR, or LS value on the RECORG keyword. This combination is not allowed because the DSNTYPE keyword only applies to non-VSAM and KSDS VSAM data sets, while the ES, RR, and LS values of the RECORG keyword only apply to VSAM data sets.

Application Programmer Action: Correct the RECORG or DSNTYPE value on the dynamic allocation request; then reissue the request.

3E82 (16002)

Meaning: The disposition field of the referencing data set was not OLD, MOD or NEW.

Application Programmer Action: Change the disposition of the referencing data set to OLD, MOD, or NEW. If the disposition was set correctly, contact the system programmer.

System Programmer Action: Locate the correct logrec data set entry for the error and report the failure to the IBM support center.

3E83 (16003)

Meaning: The disposition field of the referenced data set was not OLD, MOD or NEW.

Application Programmer Action: Change the disposition of the referenced data set to OLD, MOD, or NEW. If the disposition was set correctly, contact the system programmer.

System Programmer Action: Locate the correct logrec data set entry for the error and report the failure to the IBM support center.

3E91 (16017)

Meaning: A non-SMS-managed data set referenced an SMS-managed data set.

Application Programmer Action: Do not attempt to reference an SMS-managed data set from a non-SMS-managed data set.

3E96 (16022)

Meaning: The referenced data set is not cataloged.

Application Programmer Action: Correct the specified data set name. If data set name is valid, catalog the data set before attempting to reference it. Reissue the request.

3E97 (16023)

Meaning: The referencing data set is not cataloged.

Application Programmer Action: Correct the specified data set name. If data set name is valid, catalog the data set before using it to reference an SMS-managed data set. Reissue the request.

3EB5 (16053)

Meaning: The ACS storage class routine rejected the storage class that was derived from the referenced data set.

Application Programmer Action: Reference a data set that is in a valid storage class, and reissue the request, or contact the system programmer to have the same storage class assigned to the referencing data set as the referenced data set.

3EB6 (16054)

Meaning: The request failed because an ACS routine created by the installation failed.

System Programmer Action: Contact your storage administrator, and supply the reason code.

3EB9 (16057)

Meaning: A VOL=REF was done to a data set which is a generation data set (GDS) base. A VOL=REF to a GDS base is not valid as there are no volumes associated with a GDS base.

Application Programmer Action: Correct the VOL=REF so that the data set referenced is not a GDS base, or remove the VOL=REF.

4269 (17001)

Meaning: A duplicate data set name appears on the volume.

Application Programmer Action: Use a different data set name, and reissue the request. Contact the system programmer to resolve the duplicate data set names on the volume.

426A (17002)

Meaning: While trying to create a data set, DADSM indicated to SMS VTOC data set services that the VTOC or VTOC index on the volume is full. Therefore, SMS attempted to select another volume.

System Programmer Action: Reorganize the VTOC or VTOC index on the indicated volume.

426B (17003)

Meaning: An I/O error occurred on a volume while the data set was being deleted or renamed.

System Programmer Action: Use the record in the logrec data set, the return code, and the diagnostic information to determine the error. Use *z/OS DFSMSdfp Diagnosis* to determine the meaning of the DADSM historic return code and the diagnostic information.

426E (17006)

Meaning: The average block length is greater than 65535, the maximum allowable length.

Application Programmer Action: Reduce the average block length specified, and reissue the request.

4274 (17012)

Meaning: DADSM determined one of the following:

- The user is not authorized to create the data set specified
- The data set requires a discrete RACF profile, but RACF is not active.

Application Programmer Action: If the data set name was incorrectly specified, correct the name and reissue the request. Otherwise, contact the system programmer.

System Programmer Action: Use *z/OS DFSMSdfp Diagnosis* to determine the meaning of the DADSM historic return code and the DADSM diagnostic information from the corresponding logrec data set entry. If the return code and the diagnostic information indicate that the user is unauthorized to create the data set, then alter the user's RACF profile to grant authorization. Otherwise, remove the automatic data set protection characteristic from the user's profile.

4276 (17014)

Meaning: A track or cylinder space quantity was zero.

Application Programmer Action: Specify a non-zero value in tracks or cylinders for space, or specify a valid data class with space, and reissue the request.

428C (17036)

Meaning: During the allocation of a data set, the system found that the space requested exceeded the primary space available on all eligible volumes.

Application Programmer Action: Either reduce the directory space or increase the primary space, and reissue the request. If there is little or no space available to allocate the data set, contact the system programmer.

System Programmer Action: If volumes are fragmented, run DEFRAG or run DFSMShsm space management to create space for new data sets. If space is not available to be freed, add volumes to the storage group, or add an overflow storage group to the list of storage groups selected.

428D (17037)

Meaning: The DADSM installation exit rejected the request with a return code of 8.

System Programmer Action: Determine the reason for the rejection. Modification of the installation exit may be required.

428E (17038)

Meaning: The DADSM installation exit rejected the request with a return code of 4.

System Programmer Action: Determine the reason for the rejection. Modification of the installation exit may be required.

4295 (17045)

Meaning: Space was not specified for the creation of a data set.

Application Programmer Action: Specify a value for space, or specify a data class with non-zero space information, and reissue the request.

429B (17051)

Meaning: The primary space for the data set exceeds 65535 tracks.

Application Programmer Action: Decrease the primary space to less than 65535 tracks and reissue the request.

429E (17054)

Meaning: DADSM was unable to locate the data set on the volume.

Application Programmer Action: If the data set is not SMS managed, correct the volume specification and reissue the request. If it is a cataloged data set, and the catalog indicates that the data set resides on the volume, contact the system programmer.

System Programmer Action: Locate the logrec data set entry associated with the error to verify that the volume list passed to DADSM is correct.

42A0 (17056)

Meaning: The volume that was specified for the rename request already has a data set with the new name on it.

Application Programmer Action: Either delete the existing data set, or change the new name to a name other than the indicated data set.

42A1 (17057)

Meaning: The data set could not be deleted because it has not expired. Disposition processing cannot delete an unexpired data set.

Application Programmer Action: You can delete the data set by specifying the PURGE option on IEHPROGM SCRATCH or IDCAMS DELETE.

42A3 (17059)

Meaning: While trying to delete or rename the data set, the volume could not be mounted.

System Programmer Action: Locate the logrec data set entry associated with the error to determine the DADSM historic return code and the DADSM diagnostic information. If the error cannot be corrected, contact the IBM support center.

42A4 (17060)

Meaning: The data set cannot be deleted or renamed because it is currently in use.

Application Programmer Action: Retry the request later.

42A5 (17061)

Meaning: While trying to delete or rename the data set, the user lacks security authorization.

Application Programmer Action: If the user has access to the data set, contact the system programmer for assistance.

System Programmer Action: Locate the logrec data set entry associated with the error and contact the IBM support center.

42AE (17070)
Meaning: The data set specified was allocated as an extended format data set.

Application Programmer Action: No action required.

42AF (17071)
Meaning: The data set specified was not allocated as an extended format data set.

System Programmer Action: If the data set was intended to be an extended format data set, the volumes in the storage group may not support extended format allocations.

42B0 (17072)
Meaning: The extended sequential format requirement could not be met.

System Programmer Action: Determine why the extended sequential format requirement could not be met and ensure the storage groups and their volumes have the correct requirements for allocating an extended sequential format data set. Locate the logrec data set entry to determine if there are any other messages associated with this error.

42B1 (17073)
Meaning: The extended sequential format request will be allocated as a non-extended sequential format data set.

System Programmer Action: If this is acceptable, ignore the reason code. If not, determine why the extended sequential format requirement could not be met and ensure the storage groups and their volumes have the correct requirements for allocating an extended sequential format data set.

42B2 (17074)
Meaning: The list of volumes for the guaranteed space storage class is not valid.

Application Programmer Action: Correct the volume list for the data set or remove the volume list and let the system select the volumes. Reissue the request.

42B3 (17075)
Meaning: An attempt was made to allocate a data set that was not a physical sequential data set as an extended format data set on a system that only supports extended format data sets for physical sequential data sets.

Application Programmer Action: Ensure the data set to be allocated is physical sequential, and reissue the request.

42B4 (17076)
Meaning: An attempt was made to allocate a data set that was not SMS-managed as an extended format data set.

Application Programmer Action: Ensure the data set to be allocated will be an SMS-managed data set, and reissue the request.

42B8 (17080)
Meaning: The data set is not eligible to be allocated as a VSAM extended format data set, and will be allocated as a non-extended format data set.

Application Programmer Action: Determine why the data set was not allocated as a VSAM extended format data set, and make the changes necessary to reissue the request. If necessary, contact the system programmer for assistance.

42CC (17100)

Meaning: A catalog error or exceptional condition has caused the allocation attempt to fail.

Application Programmer Action: Examine the return code and reason code in the accompanying message IGD17100I, as described in “SMS reason code (S99ERSN)” on page 559.

42CD (17101)

Meaning: A duplicate data set name exists in the catalog.

Application Programmer Action: Allocate the data set using a different name and reissue the request.

42CE (17102)

Meaning: While trying to define a non-VSAM data set, a catalog error or exceptional condition caused the allocation attempt to fail.

Application Programmer Action: Examine the return code and reason code in accompanying message IGD17102I in *z/OS MVS System Messages, Vol 8 (IEF-IGD)*.

42D6 (17110)

Meaning: The referenced data set does not exist.

Application Programmer Action: Correct the data set name in the VOL=REF reference, and reissue the request.

42D8 (17112)

Meaning: The system programmer has overridden the expiration date or retention period specified for the data set using the ACS routines. The expiration date now meets the criteria specified in the management class for the data set.

Application Programmer Action: If the new expiration date is unacceptable, contact the system programmer to take a corrective action, if any.

42DE (17118)

Meaning: The data set referred to on the LIKE parameter is neither a non-VSAM data set nor a VSAM cluster name.

Application Programmer Action: Specify a valid data set name of a non-VSAM data set or a VSAM cluster name on the LIKE parameter, and reissue the request.

4308 (17160)

Meaning: The requested data set allocation is eligible for compression.

Application Programmer Action: If the data set should not be compressed, contact the system programmer for assistance.

4309 (17161)

Meaning: The requested data set allocation will not be compressed because compression services failed with a return code of 8.

System Programmer Action: Determine if compression services are available. If they are, locate the logrec data set entry associated with the error and contact the IBM support center. Otherwise, bring up compression services.

430A (17162)

Meaning: The requested data set allocation will not be compressed because compression services failed with a return code other than 8.

System Programmer Action: Determine if the data set characteristics met the criteria for compression. If so, locate the logrec data set entry associated with the error and contact the IBM support center. Otherwise, have the application programmer change the characteristics for the data set to ensure it will be eligible for compression.

430B (17163)

Meaning: The requested data set allocation will not be compressed because the data set characteristics do not meet the criteria for compression.

Application Programmer Action: If possible, change the characteristics for the data set to ensure it will be eligible for compression, and reissue the request.

430D (17165)

Meaning: A multi-volume temporary data set cannot be allocated as an extended format data set.

Application Programmer Action: The multi-volume temporary data set will be allocated as a non-extended format data set. The reason code can be ignored.

4331 (17201)

Meaning: SMS construct access services indicated that the data class, management class, storage class, or storage group construct for the data set does not exist in the active configuration.

Application Programmer Action: If the construct is explicitly specified, make sure the specification is correct and reissue the request. Otherwise, contact the system programmer.

System Programmer Action: If the construct was supplied by the ACS routines, the ACS routines may have to be modified.

4333 (17203)

Meaning: While trying to allocate the SMS-managed data set, SMS VTOC data set services could not retrieve a volume definition. In the volume list passed to SMS VTOC data set services, one or more of the volumes might be non-SMS-managed volumes.

System Programmer Action: If a volume list was explicitly specified, then correct the volume list and reissue the request. If you did not explicitly specify the volume list, then locate the logrec data set entry associated with the error and contact the IBM support center.

4335 (17205)

Meaning: The volumes specified are not in the same storage group for a guaranteed space request.

Application Programmer Action: Specify volumes within the same storage group, or remove the volumes, and reissue the request. If necessary, contact the system programmer for assistance.

4336 (17206)

Meaning: A space request for a data set failed because:

- No accessible volumes had sufficient space to satisfy the single-volume request
- Not enough accessible volumes had sufficient space to satisfy the multi-volume request
- In the guaranteed space request for the data set, the specified volume does not belong to any of the storage groups that the storage class mapped.

A volume is accessible if all of the following are true:

- The storage group that contains the volume is enabled to the system
- The volume itself is enabled to SMS
- The volume itself is online to MVS.

Application Programmer Action: Reissue the request, specifying less space than before. If this error message still occurs, contact the system programmer.

System Programmer Action: Determine which storage class and storage group were used for the request, and check the amount of available space on all volumes in the storage group. Then either force the selection of another storage class, or make more space available on the volumes within the selected storage group. If necessary, add more volumes to the selected storage group, add an overflow storage group to the selected storage group list, or move data off volumes using DFSMShsm.

4337 (17207)

Meaning: There are no volumes for which all of the following are true:

- The storage group that contains the volume is enabled to the system
- The volume itself is enabled to SMS
- The volume itself is online to MVS.

System Programmer Action: Determine which storage class and storage group were used for the request, and check the amount of available space on all volumes in the storage group. Then either force the selection of another storage class, or make more space available on the volumes within the selected storage group. If necessary, add more volumes to the selected storage group, add an overflow storage group to the selected storage group list, or move data off volumes using DFSMShsm.

433F (17215)

Meaning: Space was not specified or not derived from a data class for the creation of a VSAM data set.

Application Programmer Action: Specify a valid space value or contact the system programmer to have the ACS routines assign a data class that has the space specified, and then reissue the request.

4340 (17216)

Meaning: Unable to allocate space on a specified volume for a guaranteed space request.

Application Programmer Action: Have the system programmer free more space on the specified volume, specify another volume, or remove the volume, and reissue the request.

4341 (17217)

Meaning: During creation of a VSAM data set, SMS VTOC data set services volume selection was unable to select a volume for a guaranteed space request due to one of the following reasons:

- The volume is offline to MVS
- The volume is not enabled to SMS
- The volume does not contain adequate space
- The status of the storage group containing the volume was not enabled, quiesced nor quiesced new.

Application Programmer Action: Specify another volume, remove the volume, or get a non-guaranteed space storage class derived for the data set, then reissue the request. If the problem persists, contact the system programmer.

System Programmer Action: Make sure the volume is online, enabled, and has adequate space for the data set; and make sure the status of the storage group containing the volume is either enabled, quiesced, or quiesced new. Then have the application programmer reissue the request.

4344 (17220)

Meaning: More than 59 volumes were specified. The maximum number of volumes allowed is 59.

Application Programmer Action: Reduce the number of volumes specified, and reissue the request.

4345 (17221)

Meaning: A request was submitted for more than one volume for a temporary VSAM data set. Only one volume may be requested.

Application Programmer Action: Ensure that no more than one volume is specified for a temporary VSAM data set when you reissue the request.

4348 (17224)

Meaning: During creation of an SMS-managed VSAM data set, storage groups were not assigned by the automatic class selection.

System Programmer Action: Correct the storage group ACS routines.

4349 (17225)

Meaning: In the guaranteed space request for the data set, the specified volume does not belong to any of the storage groups that the storage class mapped.

Application Programmer Action: Specify another volume or remove the volume and let the system select the volume when the request is resubmitted. If not successful, contact the system programmer for assistance.

System Programmer Action: Determine which storage group contains the volume, and which storage classes map to that storage group. Ensure that the volume is online to MVS and enabled to SMS. Then force the selection of one of those storage classes when the request is resubmitted.

434A (17226)

Meaning: SMS volume selection for VSAM data sets has determined that no storage group contains enough volumes to satisfy the current request.

System Programmer Action: Do one of the following:

- Determine whether there is another storage group available that will contain the required number of volumes

- Try to match the number of required volumes to the number available in one of the eligible storage groups.

436C (17260)

Meaning: An attempt to allocate an SMS-managed data set failed because:

- The volume that the data set resides on is not enabled to the storage management subsystem on the system from which the request was made.
- The storage group that contains the data set's volume is not enabled to SMS.

System Programmer Action: Determine whether the volume or the storage group needs to be enabled; you can enable either by using the VARY SMS command. Then try to allocate the data set again.

436D (17261)

Meaning: SMS VTOC data set services was not able to allocate the data set because the first volume serial number in the data set's volume serial list is either blanks or null. The volume list was built incorrectly, possibly because of one of the following:

- The allocation request was for a VTOC index, VVDS, or VTOC data set, and the request did not specify a volume serial number.
- The allocation request was for a temporary data set and specified an incorrect volume reference.

Application Programmer Action: Make sure the allocation request specifies a volume serial number or a valid volume reference; then reissue the request.

4377 (17271)

Meaning: A request was made to allocate a SMS-managed, non-VSAM, non-guaranteed-space data set. The volume count specified (or derived from the data class) is greater than the number of available online volumes in any of the storage groups that were selected.

Application Programmer Action: Reduce the volume count and reissue the request, or contact the system programmer for assistance.

System Programmer Action: Add volume to the storage group, or ensure the storage group selected has a sufficient number of available online volumes.

4379 (17273)

Meaning: In an SMS VTOC data set services request involving the data set, one or more volumes were specified, but could not be selected. Then volume selection was reentered until all eligible volumes were tried. SMS may have been unable to select any volumes because:

- DADSM may not have found enough space
- The volume might not have been initialized as an SMS volume
- The dataset you are trying to allocate may already exist and may not be currently cataloged.

System Programmer Action: Do the following:

- If DADSM could not find enough space, put additional volumes online to MVS and enabled to SMS in one of the eligible storage groups and have the application programmer reissue the request.
- Check to see if the dataset you are trying to allocate already exists. Don't assume the dataset is currently cataloged.

- Verify that the volumes have been initialized as SMS volumes.

Otherwise, refer to the logrec data set entry associated with this error to determine if there are any messages that could describe this error further.

437A (17274)

Meaning: Volumes specified for a guaranteed space request do not belong to an eligible storage group allocation for the data set.

Application Programmer Action: Specify volumes that belong to an eligible storage group, and reissue the request.

437C (17276)

Meaning: A DISP=MOD request would exceed the maximum volume count of 59.

Application Programmer Action: Make sure no more than 59 volumes are allocated to the data set when the request is resubmitted.

437D (17277)

Meaning: A non-VSAM volume selection failed.

System Programmer Action: Locate the logrec data set entry associated with this error and determine if there are any other SMS messages that were logged with this failure.

4395 (17301)

Meaning: The volume (or volumes) on which the data set resides is either not online, or not enabled to the storage management subsystem at the time of a delete or rename request.

System Programmer Action: If the volume is offline, then have the operator vary it online. If the volume is disabled, then either enable it, or deny the delete or rename request.

4398 (17304)

Meaning: SMS VTOC data set services delete or rename processing was not able to delete the catalog entry for the data set; the data set is non-SMS-managed, and resides on one or more SMS-managed volumes that are not in initial status.

System Programmer Action: Use AMS ALTER to assign a storage class for the data set and attempt to delete it again.

4399 (17305)

Meaning: The volume definitions for the data set indicate that the data set resides on both SMS and non-SMS volumes. If any piece of a multi-volume data set resides on an SMS-managed volume, then all volumes on which the data set resides must be defined to the same storage group, and therefore be defined to SMS.

Application Programmer Action: Define all volumes on which the data set resides to the same storage group, and reissue the delete or rename request. If the catalog entry is bad, execute IDCAMS ALTER to remove volumes for the non-SMS-managed volumes, and delete the data set.

439F (17311)

Meaning: SMS VTOC data set services was called to delete or rename a data set and received a list of SMS-managed volumes. However, the data set is a non-SMS-managed data set, and does not reside on the SMS-managed volumes listed.

Application Programmer Action: If the volume list is incorrect, correct the volume list for the data set and reissue the delete or rename request. Otherwise, contact the system programmer for the corrective action.

43C7 (17351)

Meaning: The track or cylinder request was too large.

Application Programmer Action: Specify a smaller track or cylinder space quantity.

43C9 (17353)

Meaning: SMS-managed volumes were specified for a non-SMS request.

Application Programmer Action: Specify non-SMS-managed volumes and reissue the request.

43CD (17357)

Meaning: An attempt to reclaim a deferred roll-in generation data set (GDS) failed.

System Programmer Action: Locate the logrec data set entry associated with this error and determine the cause of the error using the logged messages, and then take the appropriate actions to correct the error.

43CF (17359)

Meaning: The request involving the data set specified a password. Passwords for the request are ignored for SMS-managed data sets.

Application Programmer Action: This reason code is returned for informational purposes; therefore, no action is required.

43D0 (17360)

Meaning: The retention period or expiration date specified for a temporary data set is ignored.

Application Programmer Action: A retention period or expiration date need not be specified for temporary data sets.

43D1 (17361)

Meaning: SMS VTOC data set services VSAM extend processing received non-SMS-managed volumes for an SMS-managed data set.

System Programmer Action: If the volumes in the catalog are in error, correct them by changing the volumes to '*'.

43D2 (17362)

Meaning: SMS VTOC data set services VSAM EOV processing issued a locate request for the data set to the catalog. The locate failed because the catalog entry for the data set was deleted from the catalog; all SMS-managed data sets must be cataloged.

System Programmer Action: Locate the logrec data set entry associated with the error to determine why the locate request failed. Then catalog the data set, and reissue the request.

43D4 (17364)

Meaning: On the request for a data set, the expiration date or retention period specified is greater than the maximum allowed for the management class that is effective for that data set. Therefore, the maximum value for the expiration date or retention period, is computed from the management class and assigned to the data set.

Application Programmer Action: If the expiration date is acceptable, no response is required. Otherwise, do one of the following:

- Ask the storage administrator to change the expiration date in the current management class before resubmitting the request.
- Assign a different management class to the data set before resubmitting the request.

43DB (17371)

Meaning: During allocation of the referenced data set, the UNIT for the selected VIO storage group is not defined to the system.

System Programmer Action: Disable the VIO storage group to the selected system, or change the unit name in the VIO storage group to match a unit defined to the system.

43F8 (17400)

Meaning: SMS VTOC data set services processing could not find the data set in the catalog. The data set was specified in a LIKE reference, but either was not cataloged, or included a PDS member name or a generation data group (GDG) relative generation number.

Application Programmer Action: Specify another data set in the LIKE reference, and reissue the request.

43FB (17403)

Meaning: SMS VTOC data set services like processing was unable to enqueue on a data set that was specified in a LIKE reference. SMS VTOC data set services was trying to enqueue on the data set to count the number of directory blocks.

Application Programmer Action: Remove the LIKE reference and specify a data class that meets the required criteria. Then reissue the request.

4402 (17410)

Meaning: The data set pointed to by the LIKE parameter does not reside on a direct access volume.

Application Programmer Action: Specify a data set that resides on a direct access volume and reissue the request.

4416 (17430)

Meaning: The data set referenced by the LIKE parameter is a VSAM data set. The data set referenced by LIKE must be a non-VSAM data set for creating a tape data set.

Application Programmer Action: Correct the reference or remove the LIKE parameter. Then reissue the request.

4417 (17431)

Meaning: An attempt was made to create a partitioned data set (PDS) by specifying DSNTYPE=PDS. The system could not determine the directory blocks quantity from any of the following:

- The directory blocks from the request
- The data class
- A model data set referenced by the LIKE keyword

Note: The directory block quantity cannot be picked up from a model data set that is a PDSE.

Application Programmer Action: Specify the directory blocks quantity by one of the specified means, and reissue the request.

4418 (17432)

Meaning: An inconsistent data set structure was found for the data set because of the following conditions:

- AMP=AMORG is specified in the request
- A new SMS-managed data set is being created
- No RECORG is available, either from the request or from the DATACLAS or from a data set referenced by the LIKE parameter.

Application Programmer Action: Specify a RECORG or have one assigned through the data class ACS routines. If this is not successful, contact the system programmer.

System Programmer Action: If the AMP=AMORG parameter is required but a RECORG cannot be provided at allocation, force this data set to be non-SMS-managed. This may require modification of ACS routines. Otherwise, provide the RECORG value by assigning a data class.

4419 (17433)

Meaning: The allocation of the tape data set failed because the LIKE parameter references a data set that has been migrated to tape.

Application Programmer Action: Change the LIKE reference to a data set that has not been migrated or remove the LIKE reference and specify a data class that meets the required criteria. Reissue the request.

4466 (17510)

Meaning: A DSNTYPE of PIPE was specified when the PATH keyword was not specified. This is not allowed.

Application Programmer Action: Add the PATH keyword, or remove the DSNTYPE of PIPE. Then reissue the request.

4467 (17511)

Meaning: A PATH keyword was expected, but was not specified.

Application Programmer Action: Add the PATH keyword. Then reissue the request.

4468 (17512)

Meaning: An invalid DSNTYPE was specified for the request. A DSNTYPE of PIPE is the only valid value when a PATH keyword is specified.

Application Programmer Action: Set the DSNTYPE to PIPE, or remove the PATH keyword.

4588 (17800)

Meaning: SMS construct access services indicated that the data class, management class, storage class, or storage group for the data set does not exist in the active configuration.

Application Programmer Action: If you explicitly specified the construct, make sure your specification is correct and reissue the request. Otherwise, if the construct was supplied by the ACS routines, contact the system programmer.

System Programmer Action: Change the ACS routines so that a valid construct is assigned.

458A (17802)

Meaning: In a guaranteed space request for a data set, the caller selected specific volumes, and selected a storage class with the guaranteed space attribute; therefore, the specific volumes must be honored. However, not all of the specified volumes are in the same storage group.

Application Programmer Action: Change the request so that all volumes specified are in the same storage group or remove all the specified volumes and have the system select the volumes; then reissue the request.

458B (17803)

Meaning: A space request for a data set failed because:

- No accessible volumes had sufficient space to satisfy the single-volume request.
- Not enough accessible volumes had sufficient space to satisfy the multi-volume request.
- In the guaranteed space request for data set dsn, the specified volume does not belong to any of the storage groups that the storage class mapped.

A volume is accessible if all of the following are true:

- The storage group that contains the volume is enabled to the system
- The volume itself is enabled to SMS
- The volume itself is online to MVS.

Application Programmer Action: Reissue the request, specifying less space than before. If you still get this error message, contact the system programmer.

System Programmer Action: Determine which storage class and storage group were used for the request, and check the amount of available space on all volumes in the storage group. Then either force the selection of another storage class, or make more space available on the volumes within the selected storage group.

458C (17804)

Meaning: There are no volumes for which all of the following are true:

- The storage group that contains the volume is enabled to the system
- The volume itself is enabled to SMS
- The volume itself is online to MVS.

System Programmer Action: Determine the status of all storage groups and volumes used for this request. You may need to enable some storage groups or bring some volumes online to MVS and enabled to SMS.

458E (17806)

Meaning: In an SMS VTOC data set services request involving the data set, one or more volumes were specified, but could not be selected. Then volume selection was reentered until all eligible volumes were tried. DADSM may not have found enough space or the volume might not have been initialized as an SMS volume.

System Programmer Action: If DADSM could not find enough space, put additional volumes online in one of the eligible storage groups, or run DFSMShsm space management cycle to provide more space on the volumes, and have the application programmer reissue the request. Otherwise, refer to the logrec data set entry associated with this error to determine if there are any messages that could describe this error further.

4590 (17808)

Meaning: The guaranteed space rules for allocating the data set have been relaxed, and the allocation will occur without enforcing the guaranteed space rules.

Application Programmer Action: No action is required.

Processing messages from dynamic allocation

Dynamic allocation indicates the outcome of an allocation request by a return code in general register 15 and a reason code in S99ERROR in the request block. Even when the return code indicates a successful allocation, the reason code may show that a low-level error occurred, one that was not serious enough to cause a failure. The reason code has a message associated with it, and programs that invoke dynamic allocation can process the reason code or the associated message.

Although using the message processing function is optional, failure to do so can result in difficulty diagnosing a dynamic allocation error.

This information does not describe techniques for processing the reason code; it only describes techniques for processing the message. Programs that elect to process the reason code can use IEFDB476 or DAIRFAIL to convert the reason code into the message. IEFDB476 and DAIRFAIL are IBM-supplied programs. IEFDB476 is described in "Using the functions of the IEFDB476 program." DAIRFAIL is an IBM-supplied program that is described in *z/OS TSO/E Programming Services*.

When you want messages to be issued to the end user, it is more efficient to request it through the S99EOPTS field of the request block extension, than to convert a reason code into a message by using IEFDB476.

Sending dynamic allocation messages to the end user

When a program invokes dynamic allocation, it normally does so in behalf of an end user. In a TSO/E environment, the end user is a TSO/E terminal. In a batch environment, the end user is the job. The message that dynamic allocation generates might be useful to the end user, and you can write programs to send the users these messages. Use the S99EIMSG and S99EWTP fields of the request block extension to specify that the system is to issue messages to the end user.

Some installations might need to write special message-sending programs. In this case, use the S99ERMSG field of the request block extension to request that the system return the message block to your program instead of being sent.

When you request the messages to be returned or issued, you can control the severity level of the returned or issued messages through the S99EMGSV field of the request block extension.

If you request that messages be returned, or you want to convert a reason code to a message, link to the IBM-supplied IEFDB476 program (as described in "Linking to the IEFDB476 program" on page 582) after dynamic allocation gives control back to your program.

Using the functions of the IEFDB476 program

IEFDB476 is the dynamic allocation error message processing program. It provides the following functions:

- Extracting and formatting messages
- Sending messages to an end user

- Freeing or retaining message blocks after a message has been extracted or sent.

You control the functions of IEFDB476 through an input parameter list that is mapped by IEFZB476. In the following list, the names in parentheses are those assigned by IEFZB476. For a complete description of the fields in this structure, see also EMPARMS and EMBUFS in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). And see Figure 68 on page 582 for an illustration.

Extracting messages (EMRETURN): This function extracts each dynamic allocation message from the system, formats it, and places it in a location that you specify. To extract, obtain an area of storage large enough to hold all the messages that you want to extract. Multiply 256 by the value returned in the S99ENMSG field of S99PARMS to calculate the amount of storage you need. When you invoke IEFDB476, you pass in the field EMBUFP the address of the storage you obtained. IEFDB476 places the extracted messages in the storage you've indicated. Then your program can process the messages as required. Your program is responsible for freeing this area after it has completed processing the extracted messages.

Sending messages (EMPUTLIN or EMWTP): This function sends the message to the end user. When you use this function, you must identify the end user that receives the message.

The EMPUTLIN field is for TSO/E users; the EMWTP field is for WTO users. If you specify both, EMWTP will override EMPUTLIN, and messages will be sent through WTO. If EMWTP is specified authorized callers may also request overriding the default descriptor and routing codes using the EMWTPCDE, EMWTPCDP, and EMWTDERT fields.

Freeing or retaining message blocks (EMKEEP): This function retains the storage area where the system keeps the message blocks. When you request message extraction or message sending, IEFDB476 will automatically free the associated dynamic allocation message blocks unless you explicitly specify otherwise. To request that message blocks be retained, set flag EMKEEP to one in the EMFUNCT field. After invoking IEFDB476 for message extraction or message sending, you must invoke IEFDB476 again to free the message blocks if, and only if, you requested that message blocks be retained by setting flag EMKEEP on. Free the message blocks by setting all of the function indicators in EMFUNCT to zero. Failure to free the message blocks will cause the message blocks to accumulate in storage, eventually leading to an out-of-storage condition.

The caller of IEFDB476 must be running with a PSW key value of zero, or with the same key value that matches the storage key of the storage that contains the message blocks. The storage key is specified or defaulted by the program that issues the original SVC 99 request.

Note: *Besides extracting messages, sending messages, and freeing storage, IEFDB476 can also convert an error reason code into the corresponding message. However, because DAIRFAIL also performs the same conversion function, older programs using DAIRFAIL for this purpose should continue to do so.*

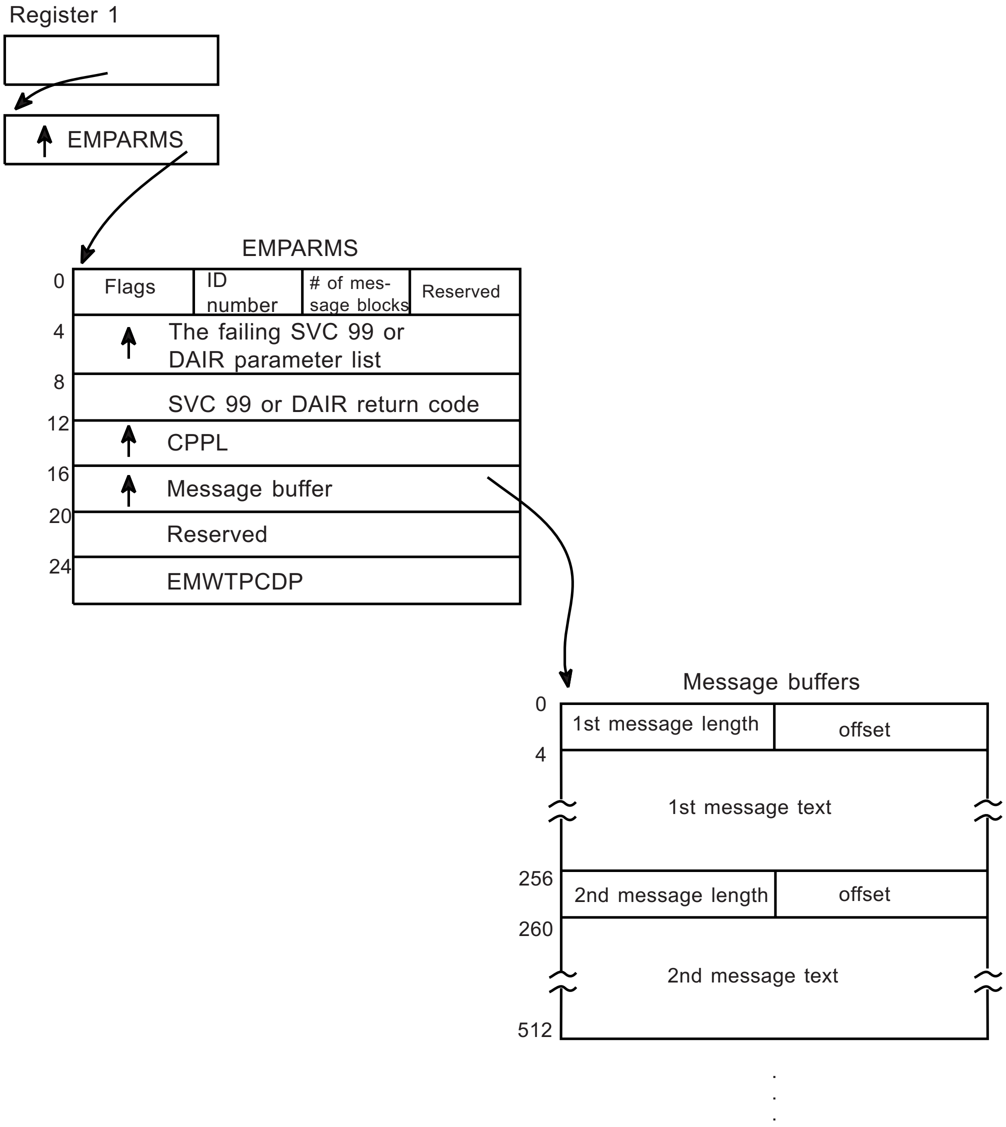


Figure 68. Structure of Input Parameter List to IEFDB476

Linking to the IEFDB476 program

Use the LINK macro to link to IEFDB476. When issuing the LINK to IEFDB476, the requirements for the caller are:

Authorization:
Problem state or supervisor state, and any PSW key

AMODE:
31-bit

RMODE:
ANY

Interrupt Status:
Enabled for I/O and external interrupts

Locks: No requirement

In addition, general purpose register 1 must contain a pointer to the error messages processing parameter list (EMPARMS) and general purpose register 13 must contain the address of a 72-byte save area.

Providing input to IEFDB476 through EMPARMS

The fields in EMPARMS control the processing of IEFDB476. Initialize the storage for EMPARMS to 0 before setting the fields to your processing options. For a complete description of EMPARMS, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Function flags (EMFUNCT): A one-byte field that identifies the functions to be performed:

Bit	Bit Name	Meaning When On
0	EMPUTLIN	Issue the messages using PUTLINE
1	EMWTP	Issue the messages using WTO to the programmer
2	EMRETURN	Return messages in the user-supplied buffer
3	EMKEEP	Do not free the storage associated with the message blocks chained out of the request block extension
4	EMWTPCDE	Authorized caller is overriding default ROUTCDE and DESC codes
5-7	EMRSV01	Not an intended programming interface

Caller identification number (EMIDNUM): A one-byte field that identifies the caller:

Value	Name	Meaning
1	EMDAIR	General caller with a DAIR error
50	EMSVC99	General caller with a DYNALLOC error
51	EMFREE	FREE command with a DYNALLOC error

Number of message blocks (EMNMSGBK): A one-byte field containing a count of the number of message blocks from which the text is to be extracted. The count of the number of message blocks that is returned from DYNALLOC is in field S99ENMSG of the request block extension. The default count is 2.

(EMRSV0x) - Not an intended programming interface: A one-byte field containing zeros.

Parameter list address (EMS99RBP): A four-byte field containing the address of the failing SVC 99 parameter list.

Parameter list address (EMDAPLP): A four-byte field containing the address of the failing DAIR parameter list.

Return code (EMRETCOD): A four-byte field containing the DYNALLOC or DAIR return code.

CPPL address (EMCPPLP): A four-byte field containing the address of the command processor parameter list. This is required only when PUTLINE is requested.

Message buffer address (EMBUFP): A four-byte field containing the address of the message buffers in which the messages are to be returned. This field is required only when bit EMRETURN in field EMFUNCT is set on.

(EMRSV0y) - Not an intended programming interface: A four-byte field containing zero.

Parameter address (EMWTPCDP): When EMWTPCDE is set, this is a four-byte address of the descriptor and route codes supplied by an authorized user to override the default DESC (7 - task related) and ROUTCDE (11 - programmer information) codes. The area can be mapped by EMWTDERT.

Parameter area (EMWTDERT) maps user-supplied descriptor and routing codes that are contiguous areas as follows:

- Descriptor codes (EMWTDESC): A 16-bit field which has a bit on for each descriptor code requested.
- Routing codes (EMWTRTCD): A 128-bit (16 byte) field which has a bit on for each routing code requested.

Not an intended programming interface: An eight-byte field containing zero.

Using EMABUFFS to receive message information from the system

The system uses the fields of the message buffer array (EMABUFFS) structure to return message information. You do not have to initialize any fields in EMABUFFS. The EMABUFFS structure is described in EMPARMS in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Length of message text (EMABUFLN): A two-byte field containing the length of the message. The length includes the lengths of EMABUFLN and EMABUFOF.

Offset (EMABUFOF): A two-byte field containing zeros.

Message text (EMABUFTX): A two-hundred fifty one byte field containing the returned message text.

Not an intended programming interface: A one-byte field containing zeroes.

Using EMBUFS to receive message information from the system

The system uses the message buffers (EMBUFS) to return message information. You do not have to initialize any fields in EMBUFS.

The EMBUFS structure is described in EMPARMS in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Length of the first message (EMBUFL1): A two-byte field containing the length of the first message. The length includes the lengths of EMBUFL1 and EMBUF01.

Offset (EMBUF01): A two-byte field containing zeros.

First message text (EMBUFT1): A two hundred fifty-one byte field containing the first returned message text.

Not an intended programming interface: A one-byte field containing zero.

Length of second message (EMBUFL2): A two-byte field containing the length of the second message returned. The length includes the length of EMBUFL2 and EMBUF02.

Offset (EMBUF02): A two-byte field containing zeroes.

Second message text (EMBUFT2): A two hundred fifty-one byte field containing the second returned message text.

Interpreting return codes from IEFDB476

IEFDB476 produces return codes in general purpose register 15. It does not produce reason codes unless the DYNALLOC caller uses a request block extension. The system returns the reason codes in the following fields of the request block extension:

- "Message processing reason code (S99ERCO)" on page 557
- "Message block freeing reason code (S99ERCF)" on page 558
- "PUTLINE/WTO macro return code (S99EWRC)" on page 558.

Hexadecimal Return Code
Meaning and Action

00 **Meaning:** The request is successful.

Application Programmer Action: None required.

04 **Meaning:** The identification number of the caller is invalid

Application Programmer Action: Correct the value supplied in the EMPARMS field EMIDNUM and resubmit the request. Valid values are described in "Caller identification number (EMIDNUM)" on page 583.

08 **Meaning:** An error occurred in PUTLINE or WTO while issuing a message. The PUTLINE or WTO return code, if any, is in the S99EWRC field of the request block extension.

Application Programmer Action: If a request block extension was supplied, the field S99ERCO contains a reason code. See "Message processing reason code (S99ERCO)" on page 557 for the appropriate action for that reason code.

If no RBX was supplied, see the descriptions of reason codes 03, 04, 06, and 07 in "Message processing reason code (S99ERCO)" on page 557 for possible causes and actions to take for this error.

0C **Meaning:** The IEFDB476 program is unable to return messages for one of the following reasons:

- A request block extension (RBX) was supplied, but the RBX field S99ENMSG contains 0, indicating that the dynamic allocation request did not return messages. The dynamic allocation request might not have returned messages because:
  - The request was successful, and no messages were returned. The RBX field S99ERCO contains a reason code of 0D.

    Application Programmer Action: See the description of reason code 0D in "Message processing reason code (S99ERCO)" on page 557 for the appropriate action.
  - o The request was unsuccessful but no messages were returned. The reason is unknown. The RBX field S99ERCO contains a reason code of 0B.

    Application Programmer Action: See the description of reason code 0B in "Message processing reason code (S99ERCO)" on page 557 for the appropriate action.
  - Bit S99ERMSG in the RBX was not set on to request that the system return messages in the message block. The RBX field S99ERCO contains a reason code of 0C.

Application Programmer Action: See the description of reason code 0C in “Message processing reason code (S99ERCO)” on page 557 for the appropriate action.

- The RBX supplied to the IEFDB476 program was not valid.

  Application Programmer Action: Verify that the RBX being supplied to IEFDB476 is the correct RBX and has not been modified.

- Message extraction was requested through bit EMRETURN in the EMFUNCT field, but no message buffer address was supplied. The RBX field S99ERCO contains a reason code of 0A.

  Application Programmer Action: See the description of reason code 0A in “Message processing reason code (S99ERCO)” on page 557 for the appropriate action.

- An RBX was supplied but the message block chain was not valid. The RBX field S99ERCO contains a reason code of 09.

  Application Programmer Action: See the description of reason code 09 in “Message processing reason code (S99ERCO)” on page 557 for the appropriate action.

10 **Meaning:** The IEFDB476 program is unable to free the storage associated with the message block chained out of the request block extension. Hexadecimal return code 10 applies only to DYNALLOC callers that have a request block extension.

Application Programmer Action: The S99ERCF field in the request block extension contains a reason code. Take the action described for that reason code in “Message block freeing reason code (S99ERCF)” on page 558.

Interpreting DYNALLOC return codes

When DYNALLOC returns control to your program, register 15 contains a return code. Depending on the return code, the S99ERROR and S99INFO fields in the input request block (S99RB) may also contain error and information reason codes. The return codes that can appear in register 15 are shown in Table 76.

The data area labels discussed in this topic are assigned by macros IEFZB4D0 and IEFZB4D2. For a complete list of the fields mapped by IEFZB4D2 and IEFZB4D0, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Table 76. DYNALLOC Return Codes

Decimal Code	Meaning and Action
0	**Meaning:** Successful completion; there will also be an information reason code returned in S99INFO if a non-terminating error occurred during request processing. **Application Programmer Action:** None required. If the system returned an information reason code, determine whether to correct it based on the needs of the application.

Table 76. DYNALLOC Return Codes (continued)

Decimal Code	Meaning and Action
4	**Meaning:** An error resulted from: • the current environment, the unavailability of a system resource (the system will also return a class 2 error reason code in S99ERROR), or a system routine failure. • the current environment (the system will also return a class 4 error reason code in S99ERROR), or • a system routine failure (the system will also return a class 7 error reason code in S99ERROR). **Application Programmer Action:** See the error reason code description and appropriate action in • Table 77 on page 592, • Table 79 on page 606, or • Table 80 on page 615 See also the associated message for the appropriate action.
8	**Meaning:** The installation validation routine denied this request. **Application Programmer Action:** Consult with your system programmer to determine whether this dynamic allocation function can be made available.
12	**Meaning:** The parameter list was not valid; the system will also return a Class 3 error reason code in S99ERROR. **Application Programmer Action:** See the Class 3 reason codes and actions listed in Table 78 on page 601, as well as any associated messages, for the appropriate action to take.

The next two topics describe the information and error reason codes that DYNALLOC returns in the request block when the return code is 0.

Interpreting information reason codes from DYNALLOC

When DYNALLOC encounters a non-terminating error during processing, the system places a return code of 0 in register 15, and an information reason code in the request block field labelled S99INFO. The information reason codes indicate that, although DYNALLOC performed the requested function, some element of the request was not processed or was overridden by system action. The actions provided with these reason codes are optional. The application programmer should determine, based on the individual program, whether it is necessary to correct the condition that caused the reason code.

Note that information reason codes (S99INFO) which accompany error reason codes (S99ERROR) in the 47zz range are actually DADSM subfunction reason codes, which are explained in *z/OS DFSMSdfp Diagnosis*. The 47zz error codes are DYNALLOC terminating errors.

The possible codes and their meanings are:

Hexadecimal Reason Code
Meaning and Action

0008 **Meaning:** The data set is deallocated using the disposition specified when the request was allocated. The overriding disposition ignored for one of the following reasons:

- The data set was originally allocated with a disposition of PASS.
- The data set is a non-subsystem data set that has a system-generated name; you cannot override disposition on this type of data set.
- The data set is a VSAM data set and the storage management subsystem (SMS) is not active.

Application Programmer Action: Determine which condition caused the code and take the appropriate action:

- Correct the overriding disposition.
- Ensure that the data set was allocated with a disposition other than PASS.
- Ensure that the data set was not a temporary data set.
- Check with your system programmer to determine if SMS is active.

Corresponding Message: IKJ68501

002n The data set was successfully deallocated but processing of the requested CATLG or UNCATLG disposition was unsuccessful. The digit “n” represents the reason for the failure, as follows:

n **Meaning and Action**

1 **Meaning:** A control volume is required; a utility program must be used to catalog the data set.

Application Programmer Action: Run the required utility program, ensuring that the required control volume is mounted.

2 **Meaning:** One of the following has occurred:

- A catalog entry already exists for the specified DSNAME.
- A catalog entry for an alias of another data set matches the specified dsname.
- The data set was (or became) a multivolume data set with a disposition of CATLG and no change was made to the volume serial list.
- The system could not locate the data set to be uncataloged.
- The dsname of a data set to be cataloged in an integrated catalog facility (ICF) catalog has the same low level qualifier (GnnnnVnn) as a GDG.
- The RACF authorization to access the data set, GDG base, or catalog was improper.
- You did not specify a password for writing the catalog, or specified it incorrectly.
- The name of the data set to be cataloged has the same high-level qualifiers as an existing alias entry in the same catalog. For example, data set A.B.C.D cannot be cataloged if an alias entry for A.B or A.B.C pointing to a different catalog already exists in the catalog.

Application Programmer Action: Correct the DSNAME and resubmit the request. If the catalog is password-protected and the system issued message IEC301A, ask your system programmer to supply the correct password to the message. Obtain authorization and the password from the system administrator or owner of the private catalog. If a RACF failure has occurred, contact the installation RACF administrator to get proper authorization.

3 **Meaning:** The specified index does not exist.

Application Programmer Action: Correct the dsname and resubmit the request.

4 **Meaning:** The data set could not be cataloged because the space was not available in the catalog.

Application Programmer Action: Increase the size of the catalog data set or delete unused catalog entries, and use a utility program to catalog the data set.

5 **Meaning:** Because too many volumes were specified for the data set, not enough storage was available to catalog it.

Application Programmer Action: Reduce the number of volumes specified, and resubmit the request.

6 **Meaning:** The data set to be cataloged in a generation index is improperly named.

Application Programmer Action: If the dsname is G0000V00:

1. Use IEHLIST to list all the data set names for the GDG
2. Use IEHPROGM to rename the data sets in the order in which they exist, starting with G0001V00 or higher. Uncatalog the data sets using the original data set names, then catalog the data sets using the new names.

If the dsname is not G0000V00, correct the dsname and resubmit the request.

7 **Meaning:** The data set to be cataloged has not been opened, and the following information was not supplied:

- *Density:* for dual density tape requests only
- *Recording mode:* for requests for which compaction information is required.

Application Programmer Action: If you want to catalog the data set, supply the required information and resubmit the request.

9 **Meaning:** An non-correctable I/O error occurred in reading or writing the catalog, or the catalog is protected by an expiration date but the purge date has not passed, or RACF denied access to the catalog.

Application Programmer Action: Resubmit the job, or catalog the data set using some alternate method, such as the IDCAMS utility. If a RACF failure has occurred, contact the installation RACF administrator to get proper authorization.

A **Meaning:** The VTOC of a DOS volume could not be converted to OS format.

Application Programmer Action: Either scratch or remove the split cylinder data set that is causing the error, and resubmit the request.

Corresponding Message: IKJ56851I

003n The data set was successfully deallocated, but processing of the requested DELETE disposition was unsuccessful. The digit “n” represents the reason for the failure, as follows:

n **Meaning**

1 **Meaning:** The expiration date has not occurred. The expiration date is specified on one of the following:

- The EXPDT or RETPD parameter of the JCL DD statement
- The EXPDT or RETPD subparameter of the JCL DD statement LABEL parameter
- The DALEXPDT or DALRETPD text unit of a previous dynamic allocation request
- The data class used for this DD statement.

Application Programmer Action: Do not attempt to delete the data set.

4 **Meaning:** No device was available for mounting the volume during deletion.

In a JES3 system, this reason code might occur if a data set was passed from one job step but not received by the job step in which it was to be deleted, as when:

- The data set was allocated to a permanently resident device that was online to MVS but off-line to JES3.
- JES3 assigned the data set to a device that was varied off-line from JES3 before the data set was deleted.

Application Programmer Action: Ensure that the correct volumes can be mounted. In a JES3 system, ensure that the device containing the data set is online to JES3.

5 **Meaning:** Because too many volumes were specified for deletion, not enough storage was available to perform the specified deletion.

Application Programmer Action: For DASD data sets, delete the VTOC entries for the data set from each volume by breaking the request up into several steps. In each step, allocate and unallocate the data set, specifying a portion of the volumes for the data set in each step. For TAPE data sets, no action is necessary.

6 **Meaning:** Either no volumes were mounted or volumes that were mounted could not be demounted to permit the remaining volumes to be mounted.

Application Programmer Action: Ensure that the correct volumes can be mounted.

8 **Meaning:** The SCRATCH routine returned an error code. If the user's JCL JOB statement requested allocation or termination messages, message IEF283I appears in the SYSOUT listing. This message lists the volume serial numbers of the data sets that were not deleted; following each number is a code that explains why each data set was not deleted.

Application Programmer Action: Take the appropriate action based on the SCRATCH error code:

Code	Action
1,5,6	Ensure that the correct volume can be mounted.
2	Supply the correct password.
3	Do not attempt to delete the data set.
4	Resubmit the request.

7 Do not specify SHR as the data set disposition.

8 Contact the RACF administrator to either properly define the data set or provide the correct RACF authorization.

9 Request the owners of the RACF-defined entities to delete the profiles from the RACF data set.

Corresponding Message: IKJ5685I. If the JOB statement requested allocation or termination messages, message IEF283I appears in the SYSOUT listing.

0044 **Meaning:** A reserved temporary data set name was used for a SYSOUT allocation. The job continued, but the system used the job name instead of the reserved name.

Application Programmer Action: Change the temporary data set name associated with the SYSOUT data set allocation.

Corresponding Message: IKJ56898I

0050 **Meaning:** The system successfully allocated the data set, but could not catalog it because the data set name is in apostrophes. The system used a disposition of KEEP instead.

Application Programmer Action: Correct the disposition, or use a dsname that is not enclosed in apostrophes.

Corresponding Message: IKJ56851I

0054 **Meaning:** The system successfully allocated the dataset. The Dynamic Allocation specified the 2-digit year Expiration Date (yyddd) Text Unit (DALEXPDT) and the 2DGT_EXPDT Policy of WARN is in effect.

A 2-digit Expiration Date year implies a year date of the form 19xx where xx is the 2-digit year specified by DALEXPDT (yyddd). Therefore, use of DALEXPDT does not allow the specification of an Expiration Date after December 31, 1999.

Application Programmer Action: Modify the Dynamic Allocation parameter input to replace the DALEXPDT text unit with the DALEXPDL text unit which uses the 4-digit year format (yyyyddd).

Corresponding Message: IEF405I

Interpreting error reason codes from DYNALLOC

When the DYNALLOC macro routines return a nonzero return code in register 15, the request block field labelled S99ERROR contains a code that explains the reason for the error.

Error reason codes are divided into the following classes:

Class	Description
1	Used for internal diagnostic purposes only. Record this code and supply it to the appropriate IBM support personnel.
2	Unavailable system resource
3	Invalid parameter list
4	Environmental error
5	Used for internal diagnostic purposes only. Record this code and supply it to the appropriate IBM support personnel.

6 Used for internal diagnostic purposes only. Record this code and supply it to the appropriate IBM support personnel.

7 System routine error

The error reason codes are shown in Table 77 through Table 80 on page 615. The class designations listed here appear as the second digit of the reason code.

Note: The explanations of the codes in these figures are followed, in parentheses, by an indication of the kind of request associated with the code.

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR)

Hex Code	(Decimal)	Meaning and Action
0204	(516)	**Meaning:** Virtual storage unavailable. **Application Programmer Action:** Resubmit the request. **Corresponding Message:** IKJ56863I
020C	(524)	**Meaning:** Unable to honor a request to upgrade an existing shared data set allocation for this address space to an exclusive allocation (such as OLD OR MOD), because the data set is also allocated to another address space, possibly on a different system in the sysplex.[1] **Application Programmer Action:** Change the allocation request and resubmit the request. **Corresponding Message:** IKJ56241I
0210	(528)	**Meaning:** Requested data set unavailable. The data set is allocated to another job and its usage attribute conflicts with this request. (dsname allocation)[1] **Application Programmer Action:** Change the allocation request and resubmit the request. **Corresponding Message:** IKJ56225I
0214	(532)	**Meaning:** Device(s) not available; or, if allocating an internal reader, all defined internal readers are already allocated. (dsname allocation)[1] **Application Programmer Action:** Ensure that the device collections of the specified device can supply the required number of devices. If necessary, change the device specification. Resubmit the request. **System Programmer Action:** If the device(s) should have been available based on the configuration defined to the system, search the problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. The S99INFO field may contain an internal informational code which can be reported to the IBM Support Center. **Corresponding Message:** IKJ56241I

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
0218	(536)	**Meaning:** Specified volume or an acceptable volume is not mounted, and user does not have volume mounting authorization through the DYNALLOC request. (dsname allocation)[1] **Application Programmer Action:** Change volume specification, or have the required volume mounted, or change the allocation request to allow volume mounting. Resubmit the request. Some possible reasons for this error are: • The dynamic allocation tried to use a VIO-eligible unit, but the data set name was not a temporary dsname (DSN=&&dsname). • The dynamic allocation tried to put a data set with a non-temporary dsname (DSN=dsname) onto a real DASD volume but – No volume serial number was supplied, and – No volumes on the system were mounted with a USE attribute of STORAGE. **Corresponding Message:** IKJ56221I
021C	(540)	**Meaning:** Device name specified is undefined. (dsname allocation) **Application Programmer Action:** Correct the device name subparameter and resubmit the allocation request. If the device name subparameter correctly identifies a device in the current configuration and a dynamic configuration change has just occurred, resubmit the request. **System Programmer Action:** If the problem recurs, search the problem reporting data bases for a fix. If none exits, contact the IBM Support Center. **Corresponding Message:** IKJ56241I
0220	(544)	**Meaning:** Requested volume not available. (dsname allocation) **Application Programmer Action:** Resubmit the request. **Corresponding Message:** IKJ56221I
0224	(548)	**Meaning:** Eligible device types do not contain enough devices. (dsname allocation)[1] **Application Programmer Action:** Change the device specification and resubmit the request. Ensure that the specified device type can supply the required number of devices. **System Programmer Action:** If the problem recurs, and the I/O configuration was built by the MVS configuration program (MVSCP), search the problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. **Corresponding Message:** IKJ56880I
0228	(552)	**Meaning:** Specified volume or device in use by system. (dsname allocation) **Application Programmer Action:** Resubmit the request when the system function has completed. **Corresponding Message:** IKJ56880I

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
022C	(556)	**Meaning:** Volume mounted on ineligible permanently resident or reserved device. (dsname allocation) **Application Programmer Action:** Ensure that the device and volume specifications are correct. If necessary, correct them. Resubmit the request. One possible reason for this error is that the dynamic allocation specified a valid volume serial number along with an invalid UNIT name. **Corresponding Message:** IKJ56221I
0230	(560)	**Meaning:** Permanently resident or reserved volume on required device. (dsname allocation) **Application Programmer Action:** Specify another device number or request the volume that is mounted on the device. **Corresponding Message:** IKJ56880I
0234	(564)	**Meaning:** More than one device required for a request specifying a specific device. (dsname allocation) **Application Programmer Action:** Change the request to specify an esoteric or generic name representing more than one device; or, if the volume is reserved and you do not wish it to be, ask the operator to unload it. Resubmit the request. **Corresponding Message:** IKJ56880I
0238	(568)	**Meaning:** Space unavailable in task input output table (TIOT). (dsname allocation, concatenation) **Application Programmer Action:** Reduce the total number of allocated DDs and devices. Deallocate data sets that are not needed simultaneously. **Corresponding Message:** IKJ56220I **Note:** The size of the TIOT increases by four (4) bytes for every SMS candidate volume per DD statement or dynamic allocation.
023C	(572)	**Meaning:** Required catalog not mounted, and user does not have volume mounting authorization. (dsname allocation) **Application Programmer Action:** Ask the operator to mount the required volume and resubmit the allocation request. **Corresponding Message:** IKJ56880I
0240	(576)	**Meaning:** Requested device is a console. (dsname allocation) **Application Programmer Action:** Change the incorrect device address and resubmit the request. **Corresponding Message:** IKJ56881I
0244	(580)	**Meaning:** Telecommunication device not accessible. (dsname allocation) **Application Programmer Action:** Ask the operator to enter VARY commands to ensure that the necessary device is accessible. Resubmit the request.
024C	(588)	**Meaning:** Operating-system-managed resource was unavailable to the subsystem. (dsname allocation)[2] **Application Programmer Action:** Consult the subsystem message. Correct the error and resubmit the allocation request.

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
0250	(592)	**Meaning:** Subsystem resource not available. (dsname allocation)[2] **Application Programmer Action:** Correct the incorrect subsystem name. Resubmit the request. **System Programmer Action:** If the subsystem name was correct, consult the subsystem documentation to determine if the subsystem supports dynamic allocation of subsystem data sets.
0254	(596)	**Meaning:** The TIOT resource is currently unavailable and the user requested conditional ENQ on the resource. (all dynamic allocation functions) **Application Programmer Action:** Resubmit the request. If acceptable, turn off the conditional ENQ request by setting bit S99CNENQ in the request block to 0.
0258	(600)	**Meaning:** There was not a sufficient number of non-restricted devices to satisfy the request, or JES3 selected a JES3-managed restricted device to satisfy the request. **Application Programmer Action:** Ensure that the device type is correctly specified. If this error code occurred because JES3 selected a device that is restricted and either JES3-managed or jointly managed, remove the device from JES3 management. **System Programmer Action:** If the problem recurs and the I/O configuration was built by the MVS configuration program (MVSCP), search the problem reporting data base for a fix for the problem. If no fix exists, contact the IBM Support Center.
025C	(604)	**Meaning:** Requested device is boxed and cannot be accessed, as a result of an I/O error condition or the operator issuing a VARY X, OFFLINE, FORCE command. (dsname allocation) **Application Programmer Action:** Resubmit the request when the device has been brought back online.
0260	(608)	**Meaning:** The device does not meet the specified status requirements. (MVS issues this reason code after a device allocation or deallocation request has failed.) **System Programmer or Operator Action:** Depending on the message accompanying this error reason code, different actions are required. Follow the actions specified in the accompanying message.
0264	(612)	**Meaning:** Due to its current status, the request made to the device is invalid. (MVS issues this reason code after a device allocation or deallocation request has failed.) **System Programmer or Operator Action:** Depending on the message accompanying this error reason code, different actions are required. Follow the actions specified in the accompanying message.
0268	(616)	**Meaning:** The tape device is not responding to request; a hardware malfunction might have occurred. (dsname allocation) **Application Programmer Action:** Change the tape device address and resubmit the request.

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
026C	(620)	**Meaning:** Request requires more SMS-managed volumes than are eligible. **Application Programmer Action:** Change the UNIT, VOLUME, or STORCLAS requirements on the allocation request to specify an eligible SMS-managed volume. **Corresponding Message:** IKJ56241I
0270	(624)	**Meaning:** Request requires more non-SMS-managed volumes than are eligible. **Application Programmer Action:** Change the UNIT, VOLUME requirements on the allocation request to specify an eligible non-SMS-managed volume. **Corresponding Message:** IKJ56241I
0274	(628)	**Meaning:** Telecommunication device deleted from I/O configuration. The accompanying message IEF358I contains the device name information. **Application Programmer Action:** Ensure that the teleprocessing device is correctly specified and is currently defined in the system configuration. **System Programmer Action:** If the error recurs, search the problem reporting data base for a fix for the problem. If no fix exists, contact the IBM Support Center. **Corresponding Message:** IEF358I
0278	(632)	**Meaning:** The requested data set is migrated, but the S99NOMIG bit is on. (S99NOMIG is described in “Flags (S99FLAG1)” on page 548) **Application Programmer Action:** Recall the data set and resubmit the request. **Corresponding Message:** IKJ56240I
0284	(644)	**Meaning:** No tape library contains enough devices to satisfy the request. The accompanying message IEF115I contains the number of devices needed. **Application Programmer Action:** Reduce the number of tape devices required and resubmit the allocation request, or, if this is not possible, notify the system programmer. **System Programmer Action:** Determine whether it is possible to satisfy the device requirements with another library. If so, assign a storage group in that library to this request. Otherwise, notify the owner of the job either to reduce the device requirements or change the request to a non-library request. **Corresponding Message:** IEF115I
028C	(652)	**Meaning:** Unable to allocate tape devices to the same generic device name. The accompanying message IEF112I contains the device name information. **Application Programmer Action:** Reduce the tape device requirements and resubmit the request. **Corresponding Message:** IEF112I

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
0290	(656)	**Meaning:** The allocation failed because the system could not assign all the required tape devices within one eligible system-managed tape library. The accompanying message IEF333I contains the libraries that are eligible to this allocation request. **Application Programmer Action:** Do one of the following: • Reduce the number of devices required by the DD statement. • Reduce the number of devices required by the step. • Balance the total number of required devices among the DD statements in the step. Then resubmit the job. **Corresponding Message:** IEF333I
0294	(660)	**Meaning:** A non-tape-library request specified a tape library device. The accompanying message IEF120I contains the device name information. **Application Programmer Action:** If you can avoid requesting a specific device (a demand request), change the device specification to either a generic device name or an esoteric device name and resubmit the allocation request. However, if you require a demand request, ensure that the required tape device is not in a tape library and resubmit the request. **Corresponding Message:** IEF120I
0298	(664)	**Meaning:** No tape device pools available for allocation. **Application Programmer Action:** Ensure that tape device pools are defined to this system and resubmit the request. **Corresponding Message:** IEF147I
029C	(668)	**Meaning:** There was a library automation communication services (LACS) mount failure for a device in a IBM 3495 Tape Library Dataserver. **System Programmer or Operator Action:** Depending on the messages accompanying this error reason code, different actions are needed. For required actions, see the accompanying message.
02A0	(672)	**Meaning:** Attempt to mount a volume on the device specified, but the mount failed. **System Programmer Action:** Depending on the messages accompanying this error reason code, different actions are needed. For required actions, see the accompanying message. **Corresponding Messages:** IEF116I, IEF118I, IEF788I, IEF789I, IEF790I, IEF791I
02A4	(676)	**Meaning:** Attempt to allocate a request to a single library failed. The accompanying message IEF333I contains the device and library name information. **Application Programmer Action:** Reduce the tape device requirements and resubmit the request. **Corresponding Message:** IEF333I

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
02A8	(680)	**Meaning:** Attempt to obtain library ID for the requested device failed. The accompanying message IEF155I contains the device name information. **System Programmer Action:** Review the logrec record related to this error for information that might help resolve why the system could not obtain the tape library ID. **Corresponding Message:** IEF155I
02AC	(684)	**Meaning:** A volume required for a non-tape-library request was mounted on library device. The accompanying message IEF151I contains the library and volume name information. **System Programmer Action:** Contact the storage administrator to check the tape volume inventory and update it as necessary. **Corresponding Message:** IEF151I
02B0	(688)	**Meaning:** A volume required for a tape-library request was mounted on a non-library device. The accompanying message IEF152I contains the library and volume names. **System Programmer Action:** Contact the storage administrator to check the tape volume inventory and update it as necessary. **Corresponding Message:** IEF152I
02B4	(692)	**Meaning:** The volume required for a tape-library request is mounted on a device in a library that was not eligible. The accompanying message IEF153I contains the library and volume name information. **System Programmer Action:** Contact the storage administrator to check the tape volume inventory and update it as necessary. **Corresponding Message:** IEF153I
02B8	(696)	**Meaning:** Unable to obtain library status for the specified tape library. The accompanying message IEF156I contains the library name information. **System Programmer Action:** Review the logrec record related to this error for information that might help to resolve why the system could not obtain the tape library name. **Corresponding Message:** IEF156I
02BC	(700)	**Meaning:** A non-tape-library request specified a library device. The accompanying message IEF113I contains the device name information. **System Programmer Action:** Choose a device that resides in the same library as the volume(s) to be allocated. **Corresponding Message:** IEF113I
02C0	(704)	**Meaning:** Unable to allocate a tape device because the device is not in the same library as the requested volume. The accompanying message IEF111I contains the device name and library name information. **System Programmer Action:** Choose a device that resides in the same library as the volume(s) to be allocated. **Corresponding Message:** IEF111I

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
02C4	(708)	**Meaning:** The allocation failed because the volume record for the specified volume could not be retrieved from the tape configuration database. The accompanying message IEF150I contains the volume serial number. **System Programmer Action:** Review the logrec record related to this error for information that might help to resolve why the system could not obtain the volume record for the specified volume serial number. **Corresponding Message:** IEF150I
02C8	(712)	**Meaning:** Request failed for a library record from the tape configuration database. **Application Programmer Action:** None. **Corresponding Message:** IEF357I
02CC	(716)	**Meaning:** The JES Client Token (CTOKEN) was not available from the currently-running JES subsystem. The usual cause of this problem is that the JES2 or JES3 version installed does not support a CTOKEN request. **System Programmer Action:** Verify that the current level of JES supports CTOKENs (which were introduced in conjunction with Client Print support). If the proper JES version is active and the problem recurs, search the problem reporting data base for a fix for the problem. If no fix exists, contact the IBM support center. The S99INFO field may contain an internal informational code which can be reported to the IBM Support Center.
02D0	(720)	**Meaning:** Allocation failed in an attempt to unload a volume needed by this job from the device where it is currently mounted. The reason for the failure is either the device ENQ failed or the device ASSIGN failed. **Corresponding Message:** IEF017I **System Programmer Action:** Resubmit the job or allocation request.
02D3	(723)	**Meaning:** Allocation failed because the eligible volume was not on an online device, and the request did not allow allocation to consider offline devices. **System Programmer Action:** Vary the eligible device online and resubmit the request.
02D4	(724)	**Meaning:** Allocation attempted to ALESERV the ALLOCAS address space, and the ALESERV service returned a non-zero return code. **System Programmer Action:** This error may indicate that the DU-AL is full. If the issuing program has issued many ALESERV ADDs and the DU-AL is full, delete some of the entries on the DU-AL and retry the request. If the error continues, search problem reporting databases for a fix for the problem. If no fix exists, contact the IBM Support Center.
02D5	(725)	**Meaning:** Allocation was already active when IEFDB440 was running as an IRB under a task. **Application Programmer Action:** IEFDB440 should not be invoked in this environment. Make sure that you adhere to this restriction. Contact the owner of the subsystem that scheduled the IRB. **System Programmer Action:** None.

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
02D6	(726)	**Meaning:** Allocation failed because there were no available offline devices within the specified esoteric, so there are no eligible devices to be brought online by Recovery Allocation processing to satisfy the request. **System Programmer Action:** Verify that the volume serial specified is valid, and that it is online in the specified esoteric. Correct the volume serial or unit information if necessary.
02D7	(727)	**Meaning:** The current environment cannot support dynamic allocation requests. **System Programmer Action:** Retry allocation request at a later time.
02D8	(728)	**Meaning:** When attempting to allocate devices to this request, Allocation was prevented from using EAV devices because of the "USEEAV(NO)" setting in the IGDSMSxx parmlib member. For a request with specific volumes in the volume list, the setting prevented the allocation. For a non-specific request, such as one with UNIT=SYSDA and no volume list, the exclusion of the EAV volumes did not leave enough volumes to allocate the request. **Application Programmer Action:** Select another esoteric unit name for the UNIT parameter, a different volume serial for the request, or contact the System Programmer. **System Programmer Action:** See System Programmer response for IEF021I. If EAV volumes are needed, use the SETSMS command to change the USEEAV setting in IGDSMSxx.
02D9	(729)	**Meaning:** Allocation failed because there were not enough available tape library devices to satisfy the request. **System Programmer Action:** If possible, make more tape devices available in the requested library by varying them available or by varying them online. Resubmit the request.
02DA	(730)	**Meaning:** The dynamic allocation request specified DALSMSHR for a tape library request, but the unit name on the DALUNIT is not valid. **Application Programmer Response:** Correct the dynamic allocation request to use a valid unit name, and resubmit the request.
02DB	(731)	**Meaning:** The dynamic allocation request specified DALSMSHR and DALUNIT for a single device, but it is a restricted device. **Application Programmer Response:** Correct the dynamic allocation request to use a device that is not restricted, and resubmit the request.
02DC	(732)	**Meaning:** The request specified DALSMSHR and DALUNIT for a single device, but the device is in use by the system. **Application Programmer Response:** Correct the error either by using another device or by waiting until the device is no longer in use by the system, and resubmit the request.
02DD	(733)	**Meaning:** The request specified DALSMSHR and DALUNIT for a single device, but the device is boxed. **Application Programmer Response:** Correct the error either by using another device or by issuing a VARY ONLINE command for the boxed device, and resubmit the request.

Table 77. Class 2 Error Reason Codes (Unavailable System Resource - ENVIRONMENTAL ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
02DE	(734)	**Meaning:** The request specified DALSMSHR and DALUNIT for a specific device, but the device is marked unavailable for allocation. **Application Programmer Response:** Correct the dynamic allocation request to use another device, or make the device available by issuing VARY ONLINE or VARY AVAILABLE commands, and resubmit the request. **Note:** The application programmers probably cannot issue the VARY commands themselves, but they must ask the system programmer to ensure that the device is online and available.
02DF	(735)	**Meaning:** The MOUNT command for a tape library device with a tape library volume specified a device which was not found in the device pools returned by the system. This could be because the media types of the device and volume requested are different. **Application Programmer Response:** Specify another device or request another volume. **Corresponding Message:** IKJ56231I

Note:

[1] The conditions that cause these return codes are detected by MVS or JES3.

[2] The information reason code contains a subsystem-defined value to further describe the error. This value is documented in publications associated with the particular subsystem.

Table 78. Class 3 Error Reason Codes (Invalid Parameter List - PROGRAM ERROR)

Hex Code	(Decimal)	Meaning and Action
0304- 0338	(772- 824)	Assigned by DAIR[4].
0358	(856)	**Meaning:** Overriding disposition of DELETE invalid for data set allocated as SHR. (unallocation)[1] **Application Programmer Action:** Do not specify a data set disposition in your input parameter list, or code a valid disposition (such as KEEP, CATALOG, or UNCATLG). If a disposition of DELETE is required, allocate the data set as OLD. **Corresponding Message:** IKJ56860I
035C	(860)	**Meaning:** Invalid PARM specified in text unit. The accompanying message IKJ56231I identifies the text unit in error. (all dynamic allocation functions)[2] **Application Programmer Action:** See the description of the text unit and correct the error. **Corresponding Message:** IKJ56231I
0360	(864)	**Meaning:** Invalid key specified in text unit. The accompanying message IKJ56231I identifies the text unit in error. (all dynamic allocation functions)[2] **Application Programmer Action:** See the description of the text unit key for the desired function and correct the error. **Corresponding Message:** IKJ56231I

Table 78. Class 3 Error Reason Codes (Invalid Parameter List - PROGRAM ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
0364	(868)	**Meaning:** JOBLIB/STEPLIB specified as a ddname, or associated with specified dsname or pathname. These ddnames are allowed only for special data sets. The accompanying message IKJ56236I identifies which of the above ddname types is in error. (dsname allocation, ddname allocation, unallocation, concatenation, deconcatenation)[1] **Application Programmer Action:** Use a different ddname, or consult your system programmer for the proper ddname to use. **Corresponding Message:** IKJ56236I
0368	(872)	**Meaning:** Authorized function requested by unauthorized user. (all dynamic allocation functions) **Application Programmer Action:** Obtain authorization from your system administrator, or request only those services (functions) that are available to unauthorized programs. **Corresponding Message:** IKJ56864I
036C	(876)	**Meaning:** Invalid parameter list format. (all dynamic allocation functions) **Application Programmer Action:** Consult the description of the DYNALLOC parameter list and make corrections. **Corresponding Message:** IKJ56231I
0374	(884)	**Meaning:** Invalid # specified in text unit. The accompanying message IKJ56231I indicates the number of the text unit in error. (all dynamic allocation functions)[2] **Application Programmer Action:** Consult the description of the text unit in error and correct the number parameter. **Corresponding Message:** IKJ56231I
0378	(888)	**Meaning:** Duplicate key specified in text unit. The accompanying message IKJ56231I indicates the number of the text unit in error. (all dynamic allocation functions)[2] **Application Programmer Action:** Remove the duplicate key from the text unit list, or substitute the intended key. **Corresponding Message:** IKJ56231I
037C	(892)	**Meaning:** Invalid LEN specified in text unit. The accompanying message IKJ56231I indicates the number of the text unit in error. (all dynamic allocation functions)[2] **Application Programmer Action:** Correct the LEN field and resubmit the request. **Corresponding Message:** IKJ56231I
0380	(896)	**Meaning:** Mutually exclusive key specified. Two keys that cannot be used together were used in the request. (dsname allocation, unallocation, information retrieval, remove-in-use processing)[2] **Application Programmer Action:** Consult the descriptions of the keys specified and determine which should be used. **Corresponding Message:** IKJ56876I

Table 78. Class 3 Error Reason Codes (Invalid Parameter List - PROGRAM ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
0384	(900)	**Meaning:** Mutually inclusive key not specified. One key was used; two should have been used. (dsname allocation, unallocation)[2] **Application Programmer Action:** Consult the description of the key in error, and the key(s) for the desired function to determine which additional key should be used. **Corresponding Message:** IKJ56877I
0388	(904)	**Meaning:** Required key not specified. (ddname allocation, information retrieval, concatenation, deconcatenation, remove-in-use processing, unallocation) **Application Programmer Action:** Consult the description of the key for the desired function to determine which key needs to be added. **Corresponding Message:** IKJ56878I
038C	(908)	**Meaning:** Duplicate ddnames specified. (concatenation) **Application Programmer Action:** Determine which ddname is a duplicate, and remove it from the list of ddnames to be concatenated (text unit 0001). For *data set* concatenation to a ddname, first allocate the data set under another ddname, then concatenate the two ddnames. **Corresponding Message:** IKJ56869I
0390	(912)	**Meaning:** GDG group name specified with relative generation number exceeds the maximum of 35 characters. (dsname allocation). **Application Programmer Action:** Change the GDG name so it does not exceed 35 characters. **Corresponding Message:** IKJ56870I
0394	(916)	**Meaning:** Status and relative generation number are incompatible. (dsname allocation) **Application Programmer Action:** Correct either the disposition specified or the relative generation level requested. **Corresponding Message:** IKJ56871I
0398	(920)	**Meaning:** Volume sequence number exceeds the number of volumes. (dsname allocation) **Application Programmer Action:** Ensure that the volume sequence number is equal to or less than the number of volume serials specified. Correct the DD specifications. **Corresponding Message**: IKJ56880I
039C	(924)	**Meaning**: Device type and volume are incompatible. (dsname allocation) **Application Programmer Action:** Correct the DD statement and resubmit the job. If the DD statement was correct, notify the system programmer. **System Programmer Action:** Verify that the I/O configuration was built correctly. If so, search problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. Check your SMS ACS routines to ensure that your non-tape and non-dasd devices have been omitted. **Corresponding Message:** IKJ56880I

Table 78. Class 3 Error Reason Codes (Invalid Parameter List - PROGRAM ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
03A0	(928)	**Meaning:** Subsystem detected an invalid parameter. (dsname allocation)[3] **Application Programmer Action:** See note [3] below. If the subsystem name was not specified (thus defaulting to the primary subsystem), check with your system programmer to obtain the name of the primary subsystem. Then consult the documentation for that subsystem and correct the parameter list error.
03A4	(932)	**Meaning:** Unable to protect data set/volume because of conflicting keyword specification. The following requirements may not have been met: • If PROTECT was specified for a DASD data set, the data set must be a non-temporary data set with a disposition of 'NEW' or 'MOD' treated as 'NEW'. If specified, normal and abnormal dispositions must be other than DELETE, and the data set has a non-temporary data set name. • If PROTECT was specified for a tape volume, the tape label specification must be SL, AL, SUL, AUL, or NSL. Both the file sequence count and volume sequence count must be set to one (except for NSL), or must default to one, and the tape volume must have a volume use attribute of PRIVATE. **Application Programmer Action:** See the description of the PROTECT specification key (0061) and remove any conflicting keys, or change the data set/volume specification to comply with the above rules. See the description of the DD statement PROTECT parameter in *z/OS MVS JCL Reference* for more information. **Corresponding Message:** IKJ56890I
03A8	(936)	**Meaning:** DYNALLOC request block extension has invalid format. **Application Programmer Action:** See the description of the DYNALLOC request block extension and correct the error. **Corresponding Message:** IKJ56231I
03AC	(940)	**Meaning:** The CPPL (command processor parameter list) address is not specified in the request block extension. This field is required if PUTLINE is used to issue messages. **Application Programmer Action:** Specify a CPPL address in the request block extension. **Corresponding Message:** IKJ56231I
03B0	(944)	**Meaning:** The overriding disposition key specified is not compatible with the file type of the specified DD. **Application Programmer Action:** If the DD represents a z/OS UNIX file, use the overriding path disposition (DUNOVPDS) key, otherwise use the normal overriding disposition (DUNOVDSP) key. **Corresponding Message:** IKJ56233I
03B4	(948)	**Meaning:** Override affinity (DALOVAFF) was specified, but a SYSOUT program name (DALSPGNM) of 'INTRDR' was not specified. **Application Programmer Action:** Specify a program name for the internal reader. **Corresponding Message:** None.

Table 78. Class 3 Error Reason Codes (Invalid Parameter List - PROGRAM ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
03B8	(952)	**Meaning:** The system does not allocate the data set and fails the dynamic allocation that specified a two-digit year expiration Date (yyddd) Text Unit (DALEXPDT) and the 2DGT_EXPDT Policy of Fail is in effect. A two-digit Expiration Date year implies a year date of the form 19xx where xx is the two-digit year specified by DALEXPDT (yyddd). Therefore, the use of DALEXPDT does not allow the specification of an Expiration Date after December 31, 1999. **Application Programmer Action:** Modify the application's dynamic allocation parameter input to replace the DALEXPDT text unit with the DALEXDPL text unit which uses the 4-digit year format. **Corresponding Message:** IEF406I.
03BC	(956)	**Meaning:** The system does not allocate the data set and fails the dynamic allocation request. The request was for a new VSAM generation data set, which is not valid because generation data sets cannot be VSAM. **Application Programmer Action:** Correct the application's dynamic allocation input to either create a generation data set that is not a VSAM data set, or a VSAM data set that is not a member of a generation data group, depending on the intent of the original allocation request. The data set name and relative generation number are indicated by the DALDSNAM and DALMEMBR text units. A data set can be explicitly defined as a VSAM data set by the DALRECO text unit, or implicitly defined by a reference to another data set or DD name using the DALDCBDS or DALDCBDD text units. **Corresponding Message:** None.
03BD	(957)	**Meaning:** The request specified DALSMSHR and DALUNIT for a non-library request. **Application Programmer Response:** Correct the dynamic allocation request to specify a different unitname for the DALUNIT text unit, or remove the DALSMSHR text unit.
03BE	(958)	**Meaning:** The request specified DALUNIT with a specific device number, DALUNCNT with a device count greater than 1, and DALSMSHR. This is not valid. **Application Programmer Response:** Correct the dynamic allocation request to remove the DALUNCNT text unit or to specify a device count of 1 on the DALUNCNT text unit.
03BF	(959)	**Meaning:** The request specified DALSMSHR and DALUNIT for a specific device, but it is not a tape device. **Application Programmer Response:** Correct the dynamic allocation request to use another device, and resubmit the request.
03C0	(960)	**Meaning:** The request specified DALSMSHR and DALUNIT for a specific device, but it is not a system-managed tape library device. **Application Programmer Response:** Correct the dynamic allocation request to use another device, and resubmit the request.

Table 78. Class 3 Error Reason Codes (Invalid Parameter List - PROGRAM ERROR) (continued)

Hex Code	(Decimal)	Meaning and Action
03C1	(961)	**Meaning:** The request specified DALSMSHR and DALUNIT for a specific device, but the device is not in the list of tape library devices selected by SMS. **Application Programmer Response:** Correct the dynamic allocation request to use another device, and resubmit the request. If the correct device was specified, consult with the system programmer to determine why SMS did not select the correct tape library for this request. **System Programmer Action:** Verify that the SMS settings (ACS routines, ISMF settings, and so on.) are returning tape library devices that are consistent with the dynamic allocation parameters, especially the DALUNIT parameter. Correct the SMS settings or help the application programmer select a more appropriate device.
03C2	(962)	**Meaning:** The request specified DALSMSHR and DALUNIT, but the unit is not an esoteric name. **Application Programmer Response:** Correct the dynamic allocation request to use an esoteric name, and resubmit the request.

Note:

[1] The information reason code field contains 0004 if the requested function was performed, although an error occurred, as the error reason code indicates.

[2] The information reason code contains the value of the key that caused the error.

[3] The information reason code field contains a subsystem-defined value to further describe the error. This value is documented in publications associated with the particular subsystem.

[4] See the description of DAIR return codes in *z/OS TSO/E Programming Services*.

Table 79. Class 4 Error Reason Codes (Environmental Error)

Hex Code	(Decimal)	Meaning and Action
0410	(1040)	**Meaning:** Specified ddname unavailable. (dsname allocation, ddname allocation) **Application Programmer Action:** Change ddname. **Corresponding Message:** IKJ56246I
0420	(1056)	**Meaning:** Specified ddname or dsname associated with an open data set. (ddname allocation, concatenation, deconcatenation, unallocation, dsname allocation)[1] **Application Programmer Action:** Change ddname or dsname, or close the open data set needed. **Corresponding Message:** IKJ56861I

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
0424	(1060)	**Meaning:** Deconcatenation would result in duplicate ddnames (deconcatenation).[1] **Application Programmer Action:** Ensure the ddname is available before resubmitting the deconcatenation request. **Corresponding Message:** IKJ56853I
0434	(1076)	**Meaning:** Ddname specified in ddname allocation request is associated with a convertible or non-permanently allocated resource. (ddname allocation) **Application Programmer Action:** Change ddname. **Corresponding Message:** IKJ56865I
0438	(1080)	**Meaning:** Specified ddname not found. (information retrieval, ddname allocation, concatenation, deconcatenation, unallocation) **Application Programmer Action:** Change ddname. **Corresponding Message:** IKJ56868I
043C	(1084)	**Meaning:** The system could not deallocate enough of the resources being held in anticipation of reuse to meet the control limit. (dsname allocation) [2] **Application Programmer Action:** Deallocate some resources or increase the control limit. **Corresponding Message:** IKJ56220I
0440	(1088)	**Meaning:** Specified data set name or pathname not found. (information retrieval, unallocation) **Application Programmer Action:** Ensure that the data set or z/OS UNIX file, for which the information was requested, exists or correct the data set name or pathname. **Corresponding Message:** IKJ56247I
0444	(1092)	**Meaning:** Relative entry number specified in information retrieval request not found. (information retrieval) **Application Programmer Action:** Change relative entry number. **Corresponding Message:** IKJ56867I
0448	(1096)	**Meaning:** Request for a new data set failed; the data set already exists. (dsname allocation) **Application Programmer Action:** Change dsname or allocate the data set as an existing data set. **Corresponding Message:** IKJ56248I
044C	(1100)	**Meaning:** Request was made for a data set that has a disposition of delete; this request cannot be honored because the data set might be deleted at any time. (dsname allocation) **Application Programmer Action:** Change the data set disposition to NEW or change the requested dsname. **Corresponding Message:** IKJ56249I

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
0450	(1104)	**Meaning:** Request caused the limit of concurrent allocations to be exceeded. (dsname allocation) • If no extended TIOT is requested, the limit is determined by the TIOT size. TIOT size is specified in the ALLOCxx member of PARMLIB, or by system default. • If extended TIOT is requested, there is no limit. **Application Programmer Action:** Deallocate data sets that are no longer needed. **Corresponding Message:** IKJ56866I
0454	(1108)	**Meaning:** Ddname in DCB reference not found. (dsname allocation) **Application Programmer Action:** Change ddname. **Corresponding Message:** IKJ56234I
0458	(1112)	**Meaning:** Dsname in DCB reference or volume reference is a GDG group name. (dsname allocation) **Application Programmer Action:** Change dsname in DCB reference to a non GDG group name. **Corresponding Message:** IKJ56879I
045C	(1116)	**Meaning:** Specified dsname to be deallocated is a member of a permanently-concatenated group. (unallocation)[1] **Application Programmer Action:** Do not attempt to deallocate by dsname a data set that is permanently concatenated. Remove the request or change it to deallocate the entire concatenation. **Corresponding Message:** IKJ56230I
0460	(1120)	**Meaning:** Specified data set name, member name, or pathname to be deallocated is not associated with specified ddname. (unallocation) **Application Programmer Action:** Change ddname, data set name, member name, or pathname. **Corresponding Message:** IKJ56872I
0464	(1124)	**Meaning:** Specified dsname to be deallocated is a private catalog. (unallocation)[1] **Application Programmer Action:** Do not attempt to deallocate a private catalog. **Corresponding Message:** IKJ56873I
0468	(1128)	**Meaning:** Error while allocating or opening a private catalog. (dsname allocation) **System Programmer Action:** List the master catalog to obtain further information about the private catalog. **Corresponding Message:** IKJ56874I

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
046C	(1132)	**Meaning:** Remote work station not defined to job entry subsystem. (dsname allocation, unallocation) **Application Programmer Action:** Change remote user ID. **System Programmer Action:** If the application programmer action does not resolve the issue, search the problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. The S99INFO field may contain an internal informational code which can be reported to the IBM Support Center. **Corresponding Message:** IKJ56875I
0470	(1136)	**Meaning:** User unauthorized for subsystem request. (dsname allocation) **System Programmer Action:** Contact the RACF administrator for assistance. If the problem recurs, search the problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. The S99INFO field may contain an internal informational code which can be reported to the IBM Support Center. **Corresponding Message:** IKJ56864I
0474	(1140)	**Meaning:** Error while attempting to select optimum device. (dsname allocation). **System Programmer Action:** If the problem recurs, search problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. **Corresponding Message:** No corresponding dynamic allocation message
0478	(1144)	**Meaning:** Unable to process job entry subsystem request. (dsname allocation, unallocation) **System Programmer Action:** If the problem recurs, search problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. The S99INFO field may contain an internal informational code which can be reported to the IBM Support Center. **Corresponding Message:** IKJ56884I
047C	(1148)	**Meaning:** Unable to establish ESTAE environment. (all dynamic allocation functions) **Application Programmer Action:** Resubmit the request. **Corresponding Message:** IKJ56231I
0480	(1152)	**Meaning:** The number of units needed to satisfy the request exceeds the limit. (dsname allocation) **Application Programmer Action:** Reduce the number of volumes needing to be mounted at the same time. An allocation for the base name of a VSAM data set includes all components. Ensure that there are less than 59 volumes requested for all components of the VSAM data set. **Corresponding Message:** IKJ56882I

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
0484	(1156)	**Meaning:** Request denied by one of the following: (dsname allocation) • The operator • The default policy specified in SYS1.PARMLIB member ALLOCxx • The installation exit specified in SYS1.PARMLIB member EXITxx. **Application Programmer Action:** Contact the system programmer for assistance. **System Programmer Action:** Verify allocation defaults and exits that were active at the time when the request was denied. If the problem recurs, search the problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. The S99INFO field may contain an internal informational code which can be reported to the IBM Support Center. **Corresponding Message:** IKJ56883I, IEF336I, IEF876I
0488	(1160)	**Meaning:** GDG pattern DSCB not mounted. (dsname allocation) **Application Programmer Action:** Ensure that the volume containing the pattern DSCB is mounted and resubmit the request. **Corresponding Message:** IKJ56880I
048C	(1164)	**Meaning:** GDG pattern DSCB not found. (dsname allocation) **Application Programmer Action:** Check the catalog volume VTOC for a DSCB for the GDG group. Correct the error and resubmit the request. **Corresponding Message:** IKJ56880I
0490	(1168)	**Meaning:** Error changing allocation assignments. (dsname allocation) **System Programmer Action:** If the problem recurs, search problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. **Corresponding Message:** No corresponding dynamic allocation message
04A4	(1188)	**Meaning:** Subsystem request in error. (dsname allocation)[3] **Application Programmer Action:** Ensure that the subsystem is installed on the processor on which the allocation request will run. **Corresponding Message:** No corresponding dynamic allocation message
04A8	(1192)	**Meaning:** Subsystem does not support allocation through key DALSSNM. (dsname allocation) **Application Programmer Action:** Do not attempt to dynamically allocate a data set for this subsystem. **Corresponding Message:** No corresponding dynamic allocation message
04AC	(1196)	**Meaning:** Subsystem is not operational. **Application Programmer Action:** Ensure that the operator makes the subsystem operational on the processor on which the allocation request will run. Resubmit the request. **Corresponding Message:** No corresponding dynamic allocation message

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
04B0	(1200)	**Meaning:** Subsystem does not exist. **Application Programmer Action:** Ensure that the subsystem is installed on the processor on which the allocation request will run. **Corresponding Message:** No corresponding dynamic allocation message
04B4	(1204)	**Meaning:** Protect request not processed; RACF not in system or not active. **Application Programmer Action:** If DASD data set or tape volume protection is required, contact the RACF administrator for assistance. **Corresponding Messages:** IJK56891I, IKJ56892I
04C0	(1216)	**Meaning:** Protect request failed; user not defined to RACF. (dsname allocation) **Application Programmer Action:** If DASD data set or tape volume protection is required, contact the RACF administrator for assistance in getting defined to RACF. **Corresponding Message:** No corresponding dynamic allocation message
04C4	(1220)	**Meaning:** The last request was for a VOL=REF to a dsname or DCB=dsname that exceeded the maximum allowable dsname backward references. (A maximum of 1489 backward references are allowed if the data set names are 44 characters in length.) **Application Programmer Action:** Reduce the number of dsname backward references. **Corresponding Message:** No corresponding dynamic allocation message
04C8	(1224)	**Meaning:** Unexpected return code encountered in internal MVS processing. This problem might have resulted from installation modification of the eligible device table (EDT). (system or installation error) **System Programmer Action:** If the installation has modified the EDT, use the ACTIVATE command or re-IPL the system to restore the EDT that existed prior to modification. If the installation has not modified the EDT, contact the IBM support center. **Corresponding Message:** IKJ56231I
04CC	(1228)	**Meaning:** Invalid output descriptor or invalid ddname reference. **System Programmer Action:** Check and correct the list of output descriptors and ddname references. Resubmit the request. **Corresponding message:** IJK56895I
04D0	(1232)	**Meaning:** SMS (Storage Management Subsystem) is not available or is at the wrong level. For allocation requests specifying pathname, the correct release of DFSMS/MVS must be installed. **Application Programmer Action:** Determine whether the data set currently resides on or is to reside on an SMS-managed volume. Contact your system programmer to find out the status of SMS on your system. **Corresponding Message:** No corresponding dynamic allocation message

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
04D4	(1236)	**Meaning:** User does not have RACF authority to allocate the specified device. **Application Programmer Action:** Request a device that you are authorized to access or notify your security administrator. **Security Administrator Action:** Determine if the user should have access to the given device. If so, define a RACF profile authorizing the user access to that device. **Corresponding message:** IJK56899I.
04D8	(1240)	**Meaning:** Unable to concatenate data sets allocated with incompatible services. **Application Programmer Action:** Do not attempt to concatenate a data set with a TIOT entry and a data set with an extended TIOT entry. **Corresponding Message:** No corresponding dynamic allocation message
04DC	(1244)	**Meaning:** The Scheduler JCL Facility (SJF) is not available. It either does not exist in the current system or the system is unable to locate it. **Application Programmer Action:** Notify the system programmer. **System Programmer Action:** Search problem reporting data bases for a fix for the problem. If no fix exists, first determine if ABEND05C RSN 0701 was issued. IF not, take a Console Dump of Master, and if an IPL is needed to resolve the problem, a Stand Alone Dump. Then contact the IBM Support Center, providing the above dump or dumps. **Corresponding Message:** No corresponding message for this error condition. Similar conditions may result in message IEF345I. The actions are the same in either case.
04E0	(1248)	**Meaning:** The actual or captured UCB attributes for the DD statements to be concatenated to not match. **Application Programmer Action:** Ensure that the actual or captured UCB attributes for the DDs being concatenated are compatible. **Corresponding Message:** No corresponding dynamic allocation message.
04E4	(1252)	**Meaning:** Unable to unload volume on an Autoswitch device due to a Coupling Facility error. **Application Programmer Action:** Rerun the job. If the error persists, notify the system programmer. **System Programmer Action:** If the system goes into automatic rebuild state, wait until the rebuild completes (indicated by message IEF268I). If the rebuild was not successful, or did not occur, take actions to correct the hardware errors indicated by the coupling facility. Once corrected, if the system does not initiate an automatic rebuild, then issue the SETXCF START,REBUILD command with the appropriate options to perform a manual rebuild of the IEFAUTOS structure. Once the rebuild is successful (indicated by message IEF268I), rerun the job. If you cannot resolve the problem, search the problem reporting data bases for a fix. If not fix exists, contact your IBM support center, providing the SYS1.LOGREC error record. **Corresponding Message:** IEF241I

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
04E5	(1253)	**Meaning:** The system cannot allocate tape devices, because a tape allocation subsystem eliminated all eligible devices from consideration. **Application Programmer Action:** Notify the system programmer. **System Programmer Action:** Determine which subsystem is causing the problem. Contact the appropriate tape subsystem vendor or service personnel. **Corresponding Message:** IEF391I
04EC	(1260)	**Meaning:** Unable to allocate an automatically switchable tape device, because the only eligible device(s) is(are) assigned to a foreign host (for example, a system not connected to the same IEFAUTOS structure, or is connected to the same structure but is using the device as a dedicated device). **Application Programmer Action:** Notify the system operator. **System Operator Action:** Determine which system has the device assigned. Issue the DISPLAY U,,, command from each system that can vary the device online. In response, message IEE457I identifies the devices assigned to that system with the letter R, (meaning reserved). If message IEE457I does not also identify the device with the letter A (meaning allocated), or one of the other status codes for message IEE457I meaning allocated to system (SYS), hardware error (BOX), busy (BSY) or other code indicating the device is unavailable, and you want to make the device available to the system on which the job or application was running, do the following: • Vary the device offline from the system it is currently assigned to. • Vary the device online to the system the job or application was running on at the time of error. • Rerun the job or restart the application that encountered the error. Otherwise, CANCEL the job or application and try again later when the device becomes availale. **Corresponding Message:** IEF284I
04F0	(1264)	**Meaning:** On a request to dynamically concatenate DD statements, the system detected that the DSABs for the requested concatenation do not all reside in the same (above-the-line or below-the-line) storage location. **Note:** All DSABs for batch allocated DD statements (JCL) reside in below-the-line storage. Dynamically allocated DD statements can request that their DSABs not reside in below-the-line storage by setting the S99DSABA or S99DXACU indicator in the SVC 99 Request Block (S99RB). **Application Programmer Action:** Ensure that the DSABs for all DDs being concatenated reside below the line or that they all reside above the line. **Corresponding Message:** IEF893I
04F4	(1268)	**Meaning:** On a request to generate a DDNAME for a dynamic allocation, the system detected that the task was at its maximum allowable limit of 65,535 for system-generated DDNAMEs. **Application Programmer Action:** Either reduce the number of concurrent DDNAME allocations or avoid using system-generated DDNAMEs. **Corresponding Message:** IKJ56227I

Table 79. Class 4 Error Reason Codes (Environmental Error) (continued)

Hex Code	(Decimal)	Meaning and Action
04F8	(1272)	**Meaning:** During a spool data set browse allocation request, JES2 determined that the data set was uninitialized (PDBMTTR was zero). **Application Programmer Action:** Resubmit the spool browse request. **System Programmer Action:** If the application programmer action does not resolve the issue, search the problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. The S99INFO field may contain an internal informational code which can be reported to the IBM Support Center. **Corresponding Message:** IEF016I
04FC	(1276)	**Meaning:** The request being processed would cause the TCT I/O table to exceed the maximum allowable size. **Application Programmer Action:** If the job that received the message has JCL DD statements that specify a high volume count, or the job that received the message uses dynamic allocation to allocate data sets and specifies a high volume count, reduce the volume count and rerun the job. If the volume count is derived from the data class, use a data class which has a lower volume count or contact the Storage Administrator. **Storage Administrator Action:** Reduce the volume count or dynamic volume count specified in the DATACLAS. **Corresponding Message:** IEF020I
04FD	(1277)	**Meaning:** Dynamic allocation request was for an esoteric that contains both TAPE and NONTAPE devices. **Application Programmer Action:** Change the request to use a different esoteric that does not contain both TAPE and NONTAPE devices and resubmit. Otherwise, notify the system programmer. **System Programmer Action:** Change the esoteric definition in the IODF so it will not mix TAPE and NONTAPE devices. ACTIVATE the new IODF and resubmit the request. **Corresponding Message:** IEF898I
04FE	(1278)	**Meaning:** Dynamic allocation request was for a tape device that was marked unavailable for allocation. **Application Programmer Action:** Change the device status using the `VARY dev,AVAILABLE` or `VARY dev,ONLINE` command and resubmit the request. Otherwise, notify the system programmer. **System Programmer Action:** Vary the device available and resubmit the request. **Corresponding Message:** IEF022I

Note:

[1] The information reason code field contains 0004 if the requested function was performed, although an error occurred as the error reason code indicates.

[2] See “Control limit” on page 539 and “Considerations when requesting dsname or pathname allocation” on page 531 for a description of these resources.

[3] The information reason code contains a subsystem-defined value to further describe the error. This value is documented in publications associated with the particular subsystem.

Table 80. Class 7 Error Reason Codes (System Routine Error)

Hex Code	Meaning and Action
17zz	CATALOG LOCATE error. LOCATE, DADSM, CATALOG, and OBTAIN errors are explained in *z/OS DFSMSdfp Advanced Services*. '08', '18', '2C', 'BA' are the only expected LOCATE return codes. 'FF' is returned as the value of zz if an unexpected return code is returned by LOCATE. (dsname allocation) The expected return codes are:
1704	**Meaning:** Unknown HSM error. **Application Programmer Action:** See message IDC3009I return code 38, reason code 4 for further explanation. **Corresponding Message:** IKJ56238I
1708	**Meaning:** One of the following occurred: • The data set name specified is in error. (program error) • A system error occurred when processing the data set name. In the request block, the S99INFO field contains the text unit key instead of an information reason code. See the corresponding message for the specific error. **Application Programmer Action:** Ensure that the correct data set name was specified. If a generation data group (GDG) level of index was coded for a non-GDG data set, remove the level of index and resubmit the job. Otherwise, this is probably a system error. Resubmit the request. If the problem persists, report the associated messages and DYNALLOC error codes to your system programmer. **System Programmer Action:** If the problem recurs and no installation action corrects the problem, search problem reporting data bases for a fix for the problem. If no fix exists, contact the appropriate IBM support personnel. **Corresponding Messages:** IKJ56228I or IKJ56229I
1718	**Meaning:** Permanent I/O error processing cataloged data set. (system error) **Application Programmer Action:** Report the messages and DYNALLOC error codes to your system programmer or operator. **System Programmer Action:** Contact hardware support. **Corresponding Message:** IKJ56231I
172C	**Meaning:** Insufficient real or virtual storage for processing cataloged data set. (environmental error) **Application Programmer Action:** Resubmit the request. If the problem persists, consult the operator to determine if another user in the installation is causing the problem, or if the entire installation is experiencing storage constraint problems. **Corresponding Message:** IKJ56863I

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
1732	**Meaning:** During a dynamic allocation, a request was made to recall a migrated data set. This recall normally runs synchronously, but the installation's storage administrator has implemented a standard patch to HSM to cause the recall to be done in the background (asynchronously). **Application Programmer Action:** Retry the dynamic allocation later. **Corresponding Message:** IKJ56242I
1734	**Meaning:** During a dynamic allocation, a request was made to recall a migrated data set. This recall normally runs synchronously, but the user converted it to a background (asynchronous) request by pressing the ATTN or PA1 keys. **TSO User Action:** Retry the dynamic allocation later. **Corresponding Message:** IKJ56243I
17BA	**Meaning:** An unsuccessful attempt was made to access a catalog that was locked for recovery. The job does not have the proper authorization to perform this function. (program error) **Application Programmer Action:** Either obtain the proper authorization, or wait until the catalog is unlocked and resubmit the request. **Corresponding Message:** IKJ56896I
17FF	**Meaning:** An unexpected return code was returned by catalog LOCATE. (system error) **Application Programmer Action:** If a GDG level of index was coded for a non-GDG data set, remove the level of index and resubmit the job. Otherwise, this is probably a system error. Report this message to the system programmer. **System Programmer Action:** If the problem recurs, search problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. **Corresponding Message:** IKJ56231I
47zz	DADSM allocate error. LOCATE, DADSM, CATALOG, and OBTAIN errors are explained in *z/OS DFSMSdfp Advanced Services*. (dsname allocation) Information reason codes which accompany error reason codes of 47zz are actually DADSM subfunction reason codes, which are explained in *z/OS DFSMSdfp Diagnosis*. The expected reason codes are as follows:
4704	**Meaning:** Duplicate data set name on the volume identified by message IKJ56229I. (program error) **Application Programmer Action:** If the specified data set is a new data set, select a unique name for it. If you intended to specify a data set that is already on the direct access device, specify OLD, SHR, or MOD in the DISP keyword. Resubmit the request. **Corresponding Message:** IKJ56229I

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
4708	**Meaning:** The volume table of contents (VTOC) for the specified or defaulted volume for a new data set request • Had insufficient space in the VTOC index for a new entry, or • Did not have the minimum number of data set control blocks (DSCBs) required to allocate the data set. (environmental error). **Application Programmer Action:** Request space on a different volume and resubmit the request. **Corresponding Message:** IKJ56229I
470C	**Meaning:** Space on the direct access storage device (DASD) containing the requested volume could not be obtained because of one of the following environmental or installation errors: • A permanent I/O error. • A incorrect format-1 data set control block (DSCB). • A structure error in the volume table of contents (VTOC) index. **Application Programmer Action:** Request space on a different volume and resubmit the request. Record the error codes and associated messages and report the problem to your operator or system programmer. **System Programmer Action:** Record the error codes, associated messages, and failing device, and report the problem to the appropriate hardware support personnel. **Corresponding Message:** IKJ56231I
4710	**Meaning:** Requested absolute track not available (environmental error). **Application Programmer Action:** Request different absolute tracks, or request non-specific space, or request space on a different volume. Resubmit the request. **Corresponding Message:** IKJ56231I
4714	**Meaning:** More space was requested than is available on the DASD volume, or the DASD volume's VTOC is full, or the DASD volume's VTOC Index (VTOCIX) is full. (environmental error) **Application Programmer Action:** Check the validity of the track quantity specified on the space quantity specification text unit keys. If it was incorrect, change it. If it was correct, change the request to a different volume. Resubmit the request. **System Programmer Action:** Enlarge the VTOC or VTOCIX as necessary. **Corresponding Message:** IKJ56245I
4718	**Meaning:** The average block length specified on text unit key DALAVGR (key 0009) was greater than the track capacity of the requested DASD volume. (environmental error) **Application Programmer Action:** Reduce the lengths of the records to make the average length no greater than the track capacity of the device, or specify a device with a greater track capacity. Resubmit the request. **Corresponding Message:** IKJ56229I
4730	**Meaning:** Invalid DADSM REALLOC parameter list. **Application Programmer Action:** Consult the *z/OS DFSMSdfp Advanced Services*, correct the parameter list, and resubmit the request.

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
4738	**Meaning:** The system did not allocate the directory for a new partitioned data set (PDS) because the space requested for the directory on text unit key DALDIR (key 000C) exceeded the space available on the specified volume. (environmental error) **Application Programmer Action:** Reduce the directory quantity specified or request a different volume. Resubmit the request. **Corresponding Message:** IKJ56244I
474C	**Meaning:** No space parameter given for a new data set or zero space requested at absolute track zero. **Application Programmer Action:** Correct the specification of space information and resubmit. **Corresponding Message:** IKJ56231I
47A8	**Meaning:** A profile for the specified data set on this volume already exists in the Resource Access Control Facility (RACF) data set. (environmental error) **Application Programmer Action:** Action may not be necessary. You may change the data set name or volume serial, or have the installation RACF administrator delete from the RACF data set the profile for the specified data set on this volume, and resubmit the request. **Corresponding Message:** IKJ56889I
47AC	**Meaning:** User not authorized to define this data set. The user has the automatic data set protection characteristic without the RACF authorization to define data sets. (program error) **Application Programmer Action:** Contact your RACF administrator to remove the automatic data set protection characteristic from the user profile, or to authorize the user to define data sets. Resubmit the request. **Corresponding Message:** IKJ56888I
47B0	**Meaning:** Installation exit rejected this request with return code 8. The information reason code (S99INFO) contains a reason code returned by the installation exit. (installation or environmental error **Application Programmer Action:** Refer to your installation procedures to determine the cause of the failure as indicated by the reason code. If appropriate, notify your system programmer of the problem. **Corresponding Message:** IKJ56222I
47B4	**Meaning:** Space request rejected by installation exit. (environmental error) **Application Programmer Action:** Refer to your installation procedures to determine the cause of the failure as indicated by the reason code that was assigned by the installation. Check the space specification text unit keys to ensure that they follow installation requirements for space requests. If they do not, correct the text unit keys and resubmit the request. If the text unit keys specify the space request correctly, notify your system programmer of the problem. **Corresponding Message:** IKJ56223I

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
47C0	**Meaning:** Non-SMS managed data set cannot be created on an SMS-managed volume. Message IKJ56897I indicates the specific operation that was rejected. (program error) **Application Programmer Action:** Change the text unit requirements and resubmit the request. **Corresponding Message:** IKJ56897I
47FF	**Meaning:** An unexpected or internal error code was received by Allocation. (system, installation, or environmental error) **Application Programmer Action:** Note the information reason code (S99INFO). If it appears in the list of DADSM CREATE Return Codes in *z/OS DFSMSdfp Diagnosis*, then take appropriate action for that return code. If not, record the error codes and associated messages and report the problem to your system programmer. **Corresponding Message:** IKJ56231I
57zz	CATALOG error. LOCATE, DADSM, CATALOG, and OBTAIN errors are explained in *z/OS DFSMSdfp Advanced Services*. (dsname allocation) The specific meanings for expected CATALOG return codes are as follows:
5704	**Meaning:** Required CATALOG volume is currently not mounted. (environmental error) **Application Programmer Action:** If possible, use another catalog. Otherwise, check with your system administrator or operator to see if the volume containing this catalog can be mounted. **Corresponding Message:** IKJ56880I
5708	**Meaning:** Either the existing catalog structure is inconsistent with the operation, or the program was not authorized to perform the operation. **Application Programmer Action:** Consult the *z/OS DFSMSdfp Advanced Services*, correct the error, and resubmit the request.
5710	**Meaning:** The index structure necessary to catalog the data set does not exist. **Application Programmer Action:** Consult the *z/OS DFSMSdfp Advanced Services*, correct the error, and resubmit the request.
5714	**Meaning:** CATALOG space is exhausted on the control volume. (environmental error) **Application Programmer Action:** If possible, use another catalog. Otherwise, record the error codes and associated messages and report the error to your system programmer. **Corresponding Message:** IKJ56231I
5718	**Meaning:** A CATALOG I/O error has occurred (system error). **Application Programmer Action:** Resubmit the request. If the problem persists, record the error codes and associated message(s) and report the error to your system programmer. **System Programmer Action:** Record the error codes, associated messages, and failing device, and report the problem to the appropriate hardware support personnel. **Corresponding Message:** IKJ56231I

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
571C	**Meaning:** One of the following occurred: • An error was found in a parameter list • There was a non-zero return code from ESTAE or GETMAIN • A permanent I/O or unrecoverable error was encountered. **Application Programmer Action:** Consult the *z/OS DFSMSdfp Advanced Services*, to correct an error in the CATALOG parameter list, and resubmit the request. For actions to take for a non-zero return code from ESTAE or GETMAIN, consult *z/OS MVS Programming: Assembler Services Reference ABE-HSP* (for unauthorized programs), or *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for authorized programs. For a permanent I/O or unrecoverable error, contact your system programmer and supply the reason code.
67zz	OBTAIN error. LOCATE, DADSM, CATALOG, and OBTAIN errors are explained in *z/OS DFSMSdfp Advanced Services*. (dsname allocation, information retrieval) The expected OBTAIN return codes are:
6708	**Meaning:** Data set not on volume as indicated in the catalog or volume text unit (installation or system error). See message IKJ56232I for more information. **Application Programmer Action:** Delete the catalog entry (use DELETE command with NOSCRATCH keyword) and recreate the data set if necessary. **Corresponding Message:** IKJ56232I
670C	**Meaning:** OBTAIN I/O error During data set allocation, an uncorrectable I/O error occurred when the system attempted to obtain a data set control block (DSCB). (system error) **Application Programmer Action:** Resubmit the request. If the problem persists, record the error codes and associated messages and report the error to your system programmer. **System Programmer Action:** Record the error codes, associated message(s) and failing device and report the problem to the appropriate hardware support personnel. **Corresponding Message:** IKJ56231I
6710	**Meaning:** An invalid work area pointer was supplied. **Application Programmer Action:** Consult the *z/OS DFSMSdfp Advanced Services*, correct the work area pointer, and resubmit the request.
6714	**Meaning:** The absolute address passed was not within the boundaries of the VTOC. **Application Programmer Action:** Consult the *z/OS DFSMSdfp Advanced Services*, correct the absolute address, and resubmit the request.
7700	**Meaning:** Subsystem error. (dsname allocation) **Application Programmer Action:** Consult the publications associated with the particular subsystem for the meaning of the subsystem-defined value, and take the appropriate action.

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
7704	**Meaning:** A subsystem interface system error occurred while processing a request for allocation of a subsystem data set (text key DALSSNM - key 005F). **Application Programmer Action:** Resubmit the request. If the problem persists, consult the subsystem documentation and installation documentation to ensure both the subsystem name and data set are supported in your installation. If the request is correct, record the name of the requested subsystem and subsystem data set, error code and any associated messages and report the problem to your system programmer. **System Programmer Action:** Ensure that the requested subsystem and subsystem data set are correct and available. If so, supply the subsystem name, data set name, error code, and associated messages to the appropriate IBM support personnel. **Corresponding Message:** IKJ56231I
8700	**Meaning:** Scheduler JCL Facility (SJF) error. This may be due to a system failure due to insufficient storage when allocating a VIO data set. (environmental or system error) **Application Programmer Action:** Resubmit the request. If the problem persists, report the DYNALLOC error code and any associated messages to your system programmer, who can supply it to the appropriate IBM support personnel. **Corresponding Message:** No corresponding dynamic allocation message
8704	**Meaning:** Scheduler JCL Facility access function error. (system error) **Application Programmer Action:** Resubmit the request. If the problem persists, report the DYNALLOC error code, information reason code, and any associated message(s) to your system programmer, who can supply it to the appropriate IBM support personnel. **Corresponding Message:** No corresponding dynamic allocation message
8708	**Meaning:** Mutual exclusivity checker error.(system error) **Application Programmer Action:** Resubmit the request. If the problem persists, report the DYNALLOC error code, information reason code, and any associated messages to your system programmer, who can supply it to the appropriate IBM support personnel. **Corresponding Message:** No corresponding dynamic allocation message
870C	**Meaning:** Scheduler JCL facility (SJF) information retrieval service error, for which there is no corresponding specific dynamic allocation reason code. **Application Programmer Action:** Resubmit the request. If the problem persists, report the DYNALLOC error code and information code and any associated messages to your system programmer, who can supply it to the IBM Support Center. **Corresponding Message:** No corresponding dynamic allocation message

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
9700	**Meaning:** Severe storage management subsystem (SMS) IDAX, ACS and Construct Access Services error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. A non-zero value in S99ERSN is a reason code. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request. If you still cannot obtain messages, contact your system programmer for assistance. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I (messages not displayed).
9704	**Meaning:** Severe SMS CATALOG service error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. A non-zero value in S99ERSN is a reason code. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request. If you still cannot obtain messages, contact your system programmer for assistance. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I (messages not displayed).
9708	**Meaning:** Severe SMS VOLREF service error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. A non-zero value in S99ERSN is a reason code. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request. If you still cannot obtain messages, contact your system programmer for assistance. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I (messages not displayed).
970C	**Meaning:** Severe SMS VTOC service error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. A non-zero value in S99ERSN is a reason code. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request. Because SMS creates entries in the logrec data set, if you still cannot obtain messages, locate the logrec data set entry for the error that corresponds to the reason code. If you need further assistance contact your system programmer. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I (messages not displayed).

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
9710	**Meaning:** Severe SMS DISP service error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. A non-zero value in S99ERSN is a reason code. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request. If you still cannot obtain messages, contact your system programmer for assistance. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I (messages not displayed).
9714	**Meaning:** Severe SMS COPY SWB service error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. A non-zero value in S99ERSN is a reason code. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request. If you still cannot obtain messages, contact your system programmer for assistance. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I (messages not displayed).
9718	**Meaning:** Error during SMS UNITAFF processing. **Application Programmer Action:** Check field S99ERSN for a non-zero value. A non-zero in S99ERSN is a reason code. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system message IKJ56894I, no messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request. If you still cannot obtain messages, contact your system programmer for assistance. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I
971C	**Meaning:** Severe SMS dataset stacking error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. See “SMS reason code (S99ERSN)” on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no SMS messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request since SMS creates entries in the logrec data set. If you still cannot obtain messages, locate the logrec data set entry for the error that corresponds to the reason code. If you need assistance, contact your system programmer. **Corresponding Message:** IKJ56893I (messages were displayed) or IKJ56894I.

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
9720	**Meaning:** Severe SMS device pool service error. **Application Programmer Action:** Check field S99ERSN for a non-zero value. See "SMS reason code (S99ERSN)" on page 559 for the possible reason codes. If the system displayed message IKJ56893I, SMS messages for the error follow IKJ56893I. If the system displayed message IKJ56894I, no SMS messages were returned. Request the message processing option of dynamic allocation to obtain related messages and resubmit the request since SMS creates entries in the logrec dataset. If you still cannot obtain messages, locate the logrec dataset entry for the error that corresponds to the reason code. If you need further assistance, contact your system programmer for assistance. **Corresponding Messages:** IKJ56893I (messages were displayed) or IKJ56894I (messages not displayed).
9724	**Meaning:** The allocation request failed because SMS provided MVS Allocation with a Library Device Pool name that could not be found in the active EDT for this allocation. This condition is primarily caused during the time after a job has entered, but has not yet completed. Device Allocation and a new SMS Library Device Pool is introduced into the configuration via a Dynamic Activate. **System Programmer Action:** Ensure that there are no outstanding configuration Activates. If there are none, search problem reporting databases for a fix for the problem. If no fix is found, contact the IBM Support Center. **Corresponding Message:** IEF124I
9728	**Meaning:** System error while allocating a device (allocation ABEND occurred - system error). **Application Programmer Action:** Resubmit the request. If the problem persists, report the error code and any associated messages to your system programmer, who can supply it to the appropriate IBM support personnel. **Corresponding Message:** No corresponding dynamic allocation message

Table 80. Class 7 Error Reason Codes (System Routine Error) (continued)

Hex Code	Meaning and Action
A700	**Meaning:** The automatic restart manager (ARM) encountered an error retrieving a copy of the symbol table from the system on which this job originated. Some of the possible errors (return code/reason code) are: **08 / AE8** A problem occured verifying the input ASCB's storage. **08 / AF0** Data was found in one or more of the reserved fields in the input parameter list. **08 / AFC** No ARB was found through the ASCB in the CopyTable. **08 / BA4** The ASST header is not valid or the ARB does not show a table. **08 / BB0** The Area for CopyTable is not valid. **System Programmer Action:** Search problem reporting data bases for a fix for the problem. If no fix exists, contact the IBM Support Center. Provide the SYSLOG or OPERLOG from the failing system and any dumps taken by the system. **Corresponding Message:** No corresponding dynamic allocation message

SVC 99 parameter list verb codes and text units, by function

The following pages contain descriptions of each of the text units you can use in the SVC 99 parameter list that is input to the DYNALLOC macro. The text units are arranged according to the functions they request, in ascending order of their KEY values. See Figure 67 on page 545 for a general description of the text unit and text unit keys.

You request a particular dynamic allocation function by coding the appropriate verb code in the request block of the SVC 99 parameter list. The text units are grouped within verb codes; the largest group (verb code 01) is further divided into three subgroups. The verb codes and the functions they represent are listed below:

Verb Code	DYNALLOC Function
X'01'	Dsname allocation
X'02'	Unallocation
X'03'	Concatenation
X'04'	Deconcatenation
X'05'	Remove-in-use processing based on task-ID
X'06'	Ddname allocation
X'07'	Information retrieval

The mnemonics given for the text units are those assigned by mapping macro IEFZB4D2. The text unit structure is mapped by mapping macro IEFZB4D0. For a complete list of the fields mapped by IEFZB4D2 and IEFZB4D0, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

If the text unit requires that you code a parameter in the PARM field, the length that you specify in the LEN field cannot be 0, unless otherwise indicated by the text unit description. Character data must be left-justified in PARM.

Table 82 on page 631 through Table 93 on page 698 present the dynamic allocation text units in list form, introductory to the descriptions of the text units in each verb code group.

Coding a dsname allocation text unit

To request dynamic allocation by dsname, use verb code 01 and the appropriate text unit key for the dsname allocation function that you require.

A suggested approach to setting up your *dsname* text unit keys is to code the applicable JCL DD statement and then look up the text unit keys you need, by JCL DD statement parameter, in Table 81 on page 627. The table is arranged by JCL parameter, and provides the equivalent text unit mnemonic, text unit key and the topic on which the text unit is described.

Table 82 on page 631 is provided to help you translate your text units by text unit key.

Note:

1. The values you specify in the text units are in hexadecimal and EBCDIC.
2. The dsname can contain system symbols. See “Using system symbols in text units” on page 627.

For dsname allocation text units that do not have a JCL equivalent, see Table 87 on page 679 and “Non-JCL dynamic allocation functions” on page 678.

JCL DD statement parameters and equivalent text units

Use Table 81 on page 627 to convert a JCL DD statement into dsname allocation text units. This table lists JCL DD statement parameters in alphabetical order, along with the equivalent verb code 01 text unit key and mnemonic. Some JCL DD statement parameters have multiple text unit keys associated with them. In these cases, the same JCL DD statement is repeated in the table for each text unit key associated with it.

JCL statement DD DCB subparameters are listed in Table 85 on page 665, although you can code them directly on a DD statement without using the DCB parameter.

The following JCL statement DD parameters do not have equivalent text units and do not appear in the table.

- *
- AMP
- CHKPT
- DATA
- DDNAME
- DLM
- DSID

Using system symbols in text units

You can use system symbols to represent data set names, member names, and path names in text units.

You can specify system symbols in the following text units:

- Verb code 01 - Dsname Allocation Text Units
  - DALDSNAM
  - DALMEMBR
  - DALPATH
- Verb code 02 - Dynamic Unallocation Text Units
  - DUNDSNAM
  - DUNMEMBR
  - DUNPATH
- Verb code 07 - Dynamic Information Retrieval Text Units
  - DINDSNAM
  - DINPATH

For detailed information about defining and using system symbols, see *z/OS MVS Initialization and Tuning Reference*.

Table 81. JCL DD Statement Parameters and Equivalent Text Units

JCL DD Statement Parameter	Text Unit Mnemonic	Text Unit Key	Described in:
ACCODE	DALACODE	8001	“Access specification - Key = '8001'” on page 651
AVGREC	DALAVGR	8010	“Average record specification - Key = '8010'” on page 654
BLKSZLIM	DALBSLM	8022	“Block size limit specification - Key = '8022'” on page 659
BURST	DALBURST	0064	“Burst specification - Key = '0064'” on page 648
CHARS	DALCHARS	0065	“Character arrangement table specification - Key = '0065'” on page 648
CNTL	DALCNTL	8003	“CNTL specification - Key = '8003'” on page 652
COPIES=group-value	DALCOPYG	0066	“Copy groups specification - Key = '0066'” on page 648
COPIES=nnn	DALCOPYS	001D	“SYSOUT copies specification - Key = '001D'” on page 641
CCSID=nnnnn	DALDCCS	8020	“CCSID specification - Key = '8020'” on page 659
DATACLAS	DALDACL	8006	“Data class specification - Key = '8006'” on page 652
DCB=*.ddname	DALDCBDD	002D	“DCB reference to a ddname specification - Key = '002D'” on page 645
DCB=dsname	DALDCBDS	002C	“DCB reference to a dsname specification - Key = '002C'” on page 645
ddname label of DD statement	DALDDNAM	0001	“Ddname specification - Key = '0001'” on page 634
DEST=node	DALSUSER	0058	“SYSOUT remote work station specification - Key = '0058'” on page 645
DEST=userid	DALUSRID	0063	“SYSOUT user ID specification - Key = '0063'” on page 647
DISP=abnormal termination disposition	DALCDISP	0006	“Data set conditional disposition specification - Key = '0006'” on page 636
DISP=normal termination disposition	DALNDISP	0005	“Data set normal disposition specification - Key = '0005'” on page 636

Table 81. JCL DD Statement Parameters and Equivalent Text Units (continued)

JCL DD Statement Parameter	Text Unit Mnemonic	Text Unit Key	Described in:
DISP=status	DALSTATS	0004	“Data set status specification - Key = '0004'” on page 635
DSNAME=dsname	DALDSNAM	0002	“Dsname specification - Key = '0002'” on page 634

DSNAME=dsname(member)
DSNAME=dsname(generation)
	DALMEMBR	0003	“Member name specification - Key = '0003'” on page 635

DSNTYPE	DALDSNT	8012	“Data set type specification - Key = '8012'” on page 654
DSNTYPE 2nd parameter	DALDSNV	802C	“DSNTYPE version - Key = '802C'” on page 663
DUMMY	DALDUMMY	0024	“Dummy data set specification - Key = '0024'” on page 643
EATTR	DALEATT	8028	“Extended attributes specification - Key = '8028'” on page 662
EXPDT=yyddd	DALEXPDT	0022	“Expiration date specification (short form) - Key = '0022'” on page 643
EXPDT=yyyy/ddd	DALEXPDL	006D	“EXPIRATION DATE specification (long form) - Key = '006D'” on page 650
FCB=fcb-name	DALFCBIM	0025	“Forms control buffer (FCB) image identification specification - Key = '0025'” on page 643

FCB=ALIGN
FCB=VERIFY
	DALFCBAV	0026	“Form alignment and image verification specification - Key = '0026'” on page 643

FILEDATA	DALFDAT	801D	“Organization of a z/OS UNIX file - Key = '801D'” on page 659
FLASH=count	DALFCNT	0068	“Flash forms overlay count specification - Key = '0068'” on page 649
FLASH=overlay-name	DALFFORM	0067	“Flash forms overlay specification - Key = '0067'” on page 648
FREE	DALCLOSE	001C	“Unallocation at CLOSE specification - Key = '001C'” on page 641
FREEVOL	DALFRVL	8029	“FREEVOL specification - Key = '8029'” on page 662
GDGORDER	DINRGDGO	802E	“GDGORDER specification – Key = '802E'” on page 664
HOLD	DALSHOLD	0059	“SYSOUT hold queue specification - Key = '0059'” on page 646
KEYENCD1	DALKYC1	8025	“Key encode 1 specification - Key = '8025'” on page 661
KEYENCD2	DALKYC2	8026	“Key encode 2 specification - Key = '8026'” on page 661
KEYLABL1	DALKYL1	8023	“Key label 1 specification - Key = '8023'” on page 660
KEYLABL2	DALKYL2	8024	“Key label 2 specification - Key = '8024'” on page 660
KEYOFF	DALKEYO	800C	“Key offset specification - Key = '800C'” on page 653
LABEL=data set sequence number	DALDSSEQ	001F	“Data set sequence number specification - Key = '001F'” on page 642
LABEL=EXPDT=yyddd	DALEXPDT	0022	“Expiration date specification (short form) - Key = '0022'” on page 643
LABEL=EXPDT=yyyy/ddd	DALEXPDL	006D	“EXPIRATION DATE specification (long form) - Key = '006D'” on page 650

LABEL=,IN
LABEL=,OUT
	DALINOUT	0021	“Input only or output only specification - Key = '0021'” on page 642

Table 81. JCL DD Statement Parameters and Equivalent Text Units (continued)

JCL DD Statement Parameter	Text Unit Mnemonic	Text Unit Key	Described in:
LABEL=label-type	DALLABEL	001E	“Label type specification - Key = '001E'” on page 641

LABEL=,PASSWORD
LABEL=,NOPWREAD
	DALPASPR	0020	“Password protection specification - Key = '0020'” on page 642

LABEL=RETPD=nnnn	DALRETPD	0023	“Retention period specification - Key = '0023'” on page 643
LIKE	DALLIKE	800F	“Copy model specification - Key = '800F'” on page 654
LGSTREAM	DALLGST	801F	“VSAM RLS log stream specification - Key = '801F'” on page 659
MAXGENS	DALMAXG	802D	“Maximum PDSE Generation - Key = '802D'” on page 664
MGMTCLAS	DALMGCL	8005	“Management class specification - Key = '8005'” on page 652
MODIFY=module-name	DALMMOD	0069	“Copy modification module specification - Key = '0069'” on page 649
MODIFY=trc	DALMTRC	006A	“Copy module table reference specification - Key = '006A'” on page 649
OUTLIM	DALOUTLM	001B	“SYSOUT output limit specification - Key = '001B'” on page 641
OUTPUT	DALOUTPT	8002	“OUTPUT statement reference - Key = '8002'” on page 651
PATH	DALPATH	8017	“z/OS UNIX file path specification - Key = '8017'” on page 655
PATHDISP=normal termination disposition	DALPNDS	801A	“z/OS UNIX file normal disposition specification - Key = '801A'” on page 658
PATHDISP=abnormal termination disposition	DALPCDS	801B	“z/OS UNIX file abnormal disposition specification - Key = '801B'” on page 658
PATHMODE	DALPMDE	8019	“z/OS UNIX file access attributes - Key = '8019'” on page 657
PATHOPTS	DALPOPT	8018	“z/OS UNIX file options - Key = '8018'” on page 656
PROTECT	DALPROT	0061	“PROTECT specification - Key = '0061'” on page 647
QNAME	DALQNAME	0027	“QNAME specification - Key = '0027'” on page 644
RECORG	DALRECO	800B	“Record organization specification - Key = '800B'” on page 653
REFDD	DALREFD	800D	“Copy DD specification - Key = '800D'” on page 653
RETPD	DALRETPD	0023	“Retention period specification - Key = '0023'” on page 643
RLS	DALRLS	801C	“Record-level sharing specification - Key = '801C'” on page 658
SECMODEL	DALSECM	800E	“Copy profile specification - Key = '800E'” on page 653
SEGMENT	DALSEGM	8014	“Segment spin data set specification - Key = '8014'” on page 655
SPACE=TRK	DALTRK	0007	“Track space type (TRK) specification - Key = '0007'” on page 636
SPACE=CYL	DALCYL	0008	“Cylinder space type (CYL) specification - Key = '0008'” on page 636

Table 81. JCL DD Statement Parameters and Equivalent Text Units (continued)

JCL DD Statement Parameter	Text Unit Mnemonic	Text Unit Key	Described in:
SPACE=blklgth	DALBLKLN	0009	“Block length specification - Key = '0009'” on page 636
SPACE=primary-quantity	DALPRIME	000A	“Primary space quantity specification - Key = '000A'” on page 637
SPACE=secondary-quantity	DALSECND	000B	“Secondary space quantity specification - Key = '000B'” on page 637
SPACE=directory	DALDIR	000C	“Directory block specification - Key = '000C'” on page 637
SPACE=RLSE	DALRLSE	000D	“Unused space release (RLSE) specification - Key = '000D'” on page 637

SPACE=CONTIG
SPACE=MXIG
SPACE=ALX
	DALSPFRM	000E	“Format of allocated space specification - Key = '000E'” on page 638

SPACE=ROUND	DALROUND	000F	“Whole cylinder allocation (ROUND) specification - Key = '000F'” on page 638
SPACE=PRIVATE	DALPRIVT	0011	“Private volume specification - Key = '0011'” on page 638
SPIN	DALSPIN	8013	“Spin data set specification - Key = '8013'” on page 654
STORCLAS	DALSTCL	8004	“Storage class specification - Key = '8004'” on page 652
SUBSYS=subsystem-name	DALSSNM	005F	“Subsystem name request specification - Key = '005F'” on page 646
SUBSYS=subsystem-parameter	DALSSPRM	0060	“Subsystem parameter specification - Key = '0060'” on page 647
SYMLIST=(sym1,sym2,..)	DALSYML	802B	“SYMLIST on a DD statement - Key = '802B'” on page 663
SYSOUT=class	DALSYSOU	0018	“SYSOUT specification - Key = '0018'” on page 640

SYSOUT=writer-name
SYSOUT=INTRDR
	DALSPGNM	0019	“SYSOUT program name specification - Key = '0019'” on page 640

SYSOUT=INTRDR	DALOVAFF	0070	“Override job affinity - Key = '0070'” on page 650
SYSOUT=form-name	DALSFMNO	001A	“SYSOUT form number specification - Key = '001A'” on page 641
TERM	DALTERM	0028	“Terminal specification - Key = '0028'” on page 644
UCS=character-set-code	DALUCS	0029	“Universal character set (UCS) specification - Key = '0029'” on page 644
UCS=FOLD	DALUFOLD	002A	“Fold mode specification - Key = '002A'” on page 644
UCS=VERIFY	DALUVRFY	002B	“Character set image verification specification - Key = '002B'” on page 645

UNIT=device-number
UNIT=device-type
UNIT=group-name
	DALUNIT	0015	“Device description specification - Key = '0015'” on page 639

UNIT=unit-count	DALUNCNT	0016	“Device count specification - Key = '0016'” on page 640
UNIT=P	DALPARAL	0017	“Parallel mount specification - Key = '0017'” on page 640
UNIT=DEFER	DALDEFER	006C	“DEFER specification - Key = '006C'” on page 649
UNIT=SMSHONOR	DALSMSHR	0076	“SMSHONOR specification – Key = '0076'” on page 651
VOLUME=SER	DALVLSER	0010	“Volume serial specification - Key = '0010'” on page 638

Table 81. JCL DD Statement Parameters and Equivalent Text Units (continued)

JCL DD Statement Parameter	Text Unit Mnemonic	Text Unit Key	Described in:
VOLUME=volume-sequence-number	DALVLSEQ	0012	“Volume sequence number specification - Key = '0012'” on page 638
VOLUME=volume-count	DALVLCNT	0013	“Volume count specification - Key = '0013'” on page 639
VOLUME=REF=dsname	DALVLRDS	0014	“Volume reference to a dsname specification - Key = '0014'” on page 639

Note: References to previous DD statements are not supported.

Dsname allocation text units

Most of the information that can be specified on a JCL DD statement can also be specified in text units for the dsname allocation function (verb code '01'). These text units are listed in Table 82 and described on the pages that follow.

The text units that represent DCB attributes are described in “DCB attribute text units” on page 666 and listed in Table 86 on page 666. The JCL DD statement DCB parameters and their equivalent text units are described in Table 85 on page 665.

The meaning of the parameters is the same as when specified on a DD statement as described in *z/OS MVS JCL Reference*.

In addition to allocating a data set, you can request that the system return information about either a dsname or a ddname dynamic allocation request by specifying either of the following:

- Information retrieval keys greater than C000, described in “Dynamic information retrieval text units” on page 697
- The non-JCL dynamic allocation functions keys 0055-0057 and 005D, described in “Non-JCL dynamic allocation functions” on page 678, which request that the system return information about your dsname allocation request.

Table 82. Verb Code 01 (Dsname Allocation) – Text Unit Keys, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	Dsname Allocation Function
0001	DALDDNAM	Associates a ddname with an allocation request.
0002	DALDSNAM	Names the data set to be allocated.
0003	DALMEMBR	Specifies data set number or relative generation number.
0004	DALSTATS	Specifies the data set status.
0005	DALNDISP	Specifies the data set's normal disposition.
0006	DALCDISP	Specifies the data set's conditional disposition.
0007	DALTRK	Specifies the space allocation in tracks.
0008	DALCYL	Specifies the space allocation in cylinders.
0009	DALBLKLN	Specifies the average data block length.
000A	DALPRIME	Specifies a primary space quantity.
000B	DALSECND	Specifies a secondary space quantity.
000C	DALDIR	Specifies the number of PDS directory blocks.
000D	DALRLSE	Deletes unused space at data set closure.
000E	DALSPFRM	Ensures a specific allocated space format.
000F	DALROUND	Specifies space allocation in whole cylinders.
0010	DALVLSER	Specifies volume serial numbers.
0011	DALPRIVT	Specifies the private volume use attribute.
0012	DALVLSEQ	Specifies the volume sequence number processing.

Table 82. Verb Code 01 (Dsname Allocation) – Text Unit Keys, Mnemonics, and Functions (continued)

Hex Text Unit Key	Mnemonic	Dsname Allocation Function
0013	DALVLCNT	Specifies the data set's volume count.
0014	DALVLRDS	Specifies volume reference to a cataloged data set.
0015	DALUNIT	Describes the unit specification.
0016	DALUNCNT	Specifies the number of devices to be allocated.
0017	DALPARAL	Specifies parallel mounting for a data set's volumes.
0018	DALSYSOU	Specifies the SYSOUT data set and defines its class.
0019	DALSPGNM	Specifies the SYSOUT program name.
001A	DALSFMNO	Specifies the SYSOUT form number.
001B	DALOUTLM	Limits the SYSOUT data set's logical record count.
001C	DALCLOSE	Frees a data set at closure.
001D	DALCOPYS	Specifies the SYSOUT listing copies count.
001E	DALLABEL	Specifies the type of volume label.
001F	DALDSSEQ	Specifies a tape data set's relative position.
0020	DALPASPR	Password protects the created data set.
0021	DALINOUT	Specifies “input only” or “output only” data set processing.
0022	DALEXPDT	Specifies the data set's expiration date.
0023	DALRETPD	Specifies the data set's retention period.
0024	DALDUMMY	Allocates a dummy data set.
0025	DALFCBIM	Identifies the forms control buffer image.
0026	DALFCBAV	Requests operator verification of the image display or forms alignment.
0027	DALQNAME	Names a TPROCESS macro.
0028	DALTERM	Specifies a time sharing terminal as an I/O device.
0029	DALUCS	Specifies a universal character set.
002A	DALUFOLD	Specifies “fold mode” for loading the requested print chain or train.
002B	DALUVRFY	Requests operator verification of the correct print chain or train mounting.
002C	DALDCBDS	Specifies the retrieval of DCB information from a cataloged data set's label.
002D	DALDCBDD	Specifies the retrieval of DCB information from a ddname-related, currently allocated data set.
0058	DALSUSER	Specifies the destination to which the SYSOUT data set is to be routed. **Note:** Keys 0058 and 0063 are equivalent to the JCL **DEST=** parameter, where **DEST=(***dalsuser***)** or **DEST=(***dalsuser,dalusrid***)**.
0059	DALSHOLD	Specifies hold queue routing for the SYSOUT data set.
005F	DALSSNM	Requests allocation of a subsystem data set.
0060	DALSSPRM	Specifies subsystem-defined parameters for use with key DALSSNM.
0061	DALPROT	Requests that the direct access data set or the tape volume be RACF-protected.
0063	DALUSRID	Specifies the destination user ID to which the SYSOUT data set is to be routed. If used, requires that DALSUSER also be specified. **Note:** Keys 0058 and 0063 are equivalent to the JCL **DEST=** parameter, where **DEST=(***dalsuser***)** or **DEST=(***dalsuser,dalusrid***)**.
0064	DALBURST	Specifies which stacker of the 3800 Printing Subsystem is to receive the paper output.
0065	DALCHARS	Specifies the name or names of character arrangement tables for printing a data set on the 3800.
0066	DALCOPYG	Specifies how copies are to be grouped if printing is done on a 3800.
0067	DALFFORM	Specifies the forms overlay to be used on the 3800 Printing Subsystem.

Table 82. Verb Code 01 (Dsname Allocation) – Text Unit Keys, Mnemonics, and Functions (continued)

Hex Text Unit Key	Mnemonic	Dsname Allocation Function
0068	DALFCNT	Specifies the number of copies on which the forms overlay is to be printed.
0069	DALMMOD	Specifies the name of the copy modification module to be loaded into the 3800 Printing Subsystem.
006A	DALMTRC	Specifies the table reference character that corresponds to a character arrangement table used for printing the copy modification data.
006C	DALDEFER	Specifies that the system should allocate a device to the data set, but defer mounting the volume(s) until the data set is opened.
006D	DALEXPDL	Specifies the data set's expiration date. This differs from DALEXPDT because the year is specified with 4 digits instead of 2.
0070	DALOVAFF	Specifies the override of system affinity for a job submitted to the internal reader.
0071	DALRTCTK	Specifies that the JES Client Token (CTOKEN) be returned to the caller of DYNALLOC.
0076	DALSMSHR	Indicates that the system must attempt to allocate to the device or the esoteric on the unit name for an SMS tape library request.
8001	DALACODE	Specifies an access code for an IOS/ANSI/FIPS Version 3 tape data set.
8002	DALOUTPT	Refers to a specific OUTPUT JCL statement or dynamic output descriptor
8003	DALCNTL	Refers to a JCL CNTL statement.
8004	DALSTCL	Specifies the storage class of a new SMS-managed data set.
8005	DALMGCL	Specifies the management class of a new SMS-managed data set.
8006	DALDACL	Specifies the data class of a new SMS-managed data set.
800B	DALRECO	Specifies the record organization of a VSAM data set.
800C	DALKEYO	Specifies the key offset of a VSAM data set.
800D	DALREFD	Specifies the name of the JCL DD statement from which the attributes are to be copied.
800E	DALSECM	Specifies the name of the RACF profile from which the RACF profile is to be copied.
800F	DALLIKE	For SMS-managed data sets, specifies the name of a model data set from which the attributes are to be copied.
8010	DALAVGR	Specifies the allocation unit to be used when the data set is allocated.
8012	DALDSNT	Specifies a data set type attribute.
8013	DALSPIN	Specifies whether the output for the SYSOUT data set is to be printed immediately upon unallocation of the data set, or at the end of the job.
8014	DALSEGM	Specifies the number of logical, line-mode pages (the segment) to be produced for a SYSOUT data set before the segment becomes eligible for immediate printing.
8017	DALPATH	Specifies the z/OS UNIX file pathname.
8018	DALPOPT	Specifies the z/OS UNIX file options.
8019	DALPMDE	Specifies the z/OS UNIX file access attributes.
801A	DALPNDS	Specifies the disposition of the z/OS UNIX file during normal step termination.
801B	DALPCDS	Specifies the disposition of the z/OS UNIX file during abnormal step termination.
801C	DALRLS	Specifies the record-level sharing protocol for a VSAM data set.
801D	DALFDAT	Specifies the organization of a z/OS UNIX file.
801F	DALLGST	Specifies the VSAM RLS log stream.
8020	DALDCCS	Specifies the Coded Character Set Identifier.

Table 82. Verb Code 01 (Dsname Allocation) – Text Unit Keys, Mnemonics, and Functions (continued)

Hex Text Unit Key	Mnemonic	Dsname Allocation Function
8022	DALBSLM	Specifies the upward limit that is used by the system-determined block size processing.
8023	DALKYL1	Specifies the label for the key encrypting key used by the key manager. The key encrypting key is used to encrypt the data (encryption) key.
8024	DALKYL2	Specifies the label for the key encrypting key used by the key manager. The key encrypting key is used to encrypt the data (encryption) key.
8025	DALKCD1	Specifies how the label for the key encrypting key specified by DALKYL1 for this DD is to be encoded by the key manager and stored on the tape cartridge.
8026	DALKCD2	Specifies how the label for the key encrypting key specified by DALKYL2 for this DD is to be encoded by the key manager and stored on the tape cartridge.
8028	DALEATT	Indicates whether the data set can support extended attributes (format 8 and 9 DSCBs).
8029	DALFRVL	Specifies whether to allow other jobs to read freed volumes of a multivolume tape file as the volume is dismounted by the job
802A	DALSPI2	Specifies the SPIN interval for the allocated SYSOUT data set.
802B	DALSYML	Lists the symbol names to be passed to JES when an internal reader is allocated.
802C	DALDSNV	Specifies the data set type version information.
802D	DALMAXG	When you have applied APAR OA42358, specifies the maximum number of PDSE member generations to be retained by the system.
802E	GDGORDER	Specifies the order in which the individual generation data sets (GDSs) are concatenated.

Ddname specification - Key = '0001'

DALDDNAM specifies a ddname to be associated with a dsname allocation request. When you code this key, # must be one, LEN is the length of the ddname, up to a maximum of 8, and PARM contains the ddname.

Example: To specify the ddname DD1, code:

```
KEY    #     LEN   PARM
0001   0001  0003  C4 C4 F1
```

Dsname specification - Key = '0002'

DALDSNAM specifies the name of the data set to be allocated. The data set name can contain special characters, if the data set name is enclosed in apostrophes. The system cannot catalog a data set name enclosed in apostrophes; it will use a disposition of KEEP instead. The data set name can contain system symbols. See the information on using system symbols in *z/OS MVS Initialization and Tuning Reference* for more information.

Dynamic allocation does not support backward references. See Table 69 on page 527.

The QNAME (DALQNAME) and IPLTXTID (DALIPLTX) keys are mutually exclusive with DALDSNAM. When you code this key, # must be one, LEN is the length of the dsname, and PARM contains the dsname.

The maximum length of the data set name is 44 characters, excluding any enclosing apostrophes and compressing any double apostrophes within the data set name.

Example: To specify the dsname MYDATA, code:

```
KEY    #      LEN    PARM
0002   0001   0006   D4 E8 C4 C1 E3 C1
```

Example: To specify the temporary dsname &LOAD, code:

```
KEY    #      LEN    PARM
0002   0001   0005   50 D3 D6 C1 C4
```

Example: To specify the dsname A.B, code:

```
KEY    #      LEN    PARM
0002   0001   0003   C1 4B C2
```

Member name specification - Key = '0003'

DALMEMBR specifies that a particular member of a data set is to be allocated, rather than the entire data set. If the member does not exist and DISP=OLD or DISP=SHR is specified, the allocation will succeed but the job will fail when the data set is opened for input. If the member does not exist and the data set is opened for output, the system will add the member to the data set. The data set member name can contain system symbols. See the information on using system symbols in *z/OS MVS Initialization and Tuning Reference* for more information. A relative generation group number may be specified as the member name.

When you specify DALMEMBR, you must also specify the dsname key (DALDSNAM). The QNAME (DALQNAME) and IPLTXTID (DALIPLTX) keys are mutually exclusive with DALMEMBR. When you code this key, # must be one, LEN is the actual length of the member name, and PARM contains the member name.

Example: To specify the member name MEM1, code:

```
KEY    #      LEN     PARM
0003   0001   0004    D4 C5 D4 F1
```

Example: To specify the relative generation number +1, code:

```
KEY    #      LEN     PARM
0003   0001   0002    4E F1
```

Data set status specification - Key = '0004'

DALSTATS specifies the data set status desired. It is mutually exclusive with the SYSOUT key (DALSYSOU). When you code DALSTATS, # and LEN must be one, and PARM contains one of the following values:

X'01'	OLD
X'02'	MOD
X'04'	NEW
X'08'	SHR

Example: To specify a status of NEW, code:

```
Key    #      LEN    PARM
0004   0001   0001   04
```

Do not code MOD for temporary data sets dynamically allocated as &&dsname.

Data set normal disposition specification - Key = '0005'

DALNDISP specifies the normal data set disposition desired. It is mutually exclusive with the SYSOUT key (DALSYSOU). When you code DALNDISP, # and LEN must be one, and PARM contains one of the following values:

X'01'	UNCATLG
X'02'	CATLG
X'04'	DELETE
X'08'	KEEP

Example: To specify a normal disposition of DELETE, code:

```
KEY    #      LEN    PARM
0005   0001   0001   04
```

Data set conditional disposition specification - Key = '0006'

DALCDISP specifies the conditional data set disposition desired. It is mutually exclusive with the SYSOUT key (DALSYSOU). The values for #, LEN, and PARM are the same as for normal disposition.

Example: To specify a conditional disposition of DELETE, code:

```
KEY    #      LEN    PARM
0006   0001   0001   04
```

Track space type (TRK) specification - Key = '0007'

DALTRK specifies that space is to be allocated in tracks. The primary quantity space key (DALPRIME) or the secondary quantity space key (DALSECND) must also be specified when you code DALTRK. The text unit keys that define space in terms of cylinders (DALCYL, DALROUND) or blocks (DALBLKLN) are mutually exclusive with DALTRK. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify a space request in tracks, code:

```
KEY    #      LEN   PARM
0007   0000   -     -
```

Cylinder space type (CYL) specification - Key = '0008'

DALCYL specifies that space is to be allocated in cylinders. The primary quantity space key (DALPRIME) or secondary quantity space key (DALSECND) must also be specified when you code this key. The text unit keys that define space in terms of tracks (DALTRK) or blocks (DALBLKLN) are mutually exclusive with DALCYL. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify a space request in cylinders, code:

```
KEY    #      LEN   PARM
0008   0000   -     -
```

Block length specification - Key = '0009'

If you do not code DALAVGR, DALBLKLN specifies the average **data block length** the system will use when computing the amount of space to allocate.

If you do code DALAVGR (with SMS), DALBLKLN specifies the average **record length** in bytes of the data. The system computes the block size and the number of tracks to allocate.

You must also specify the primary quantity space key (DALPRIME) or the secondary quantity space key (DALSECND) when you code this key. The text unit keys that request space in terms of tracks (DALTRK) or cylinders (DALCYL, DALROUND) should not be specified with DALBLKLN. When you code this key, # must be one, LEN must be three, and PARM contains the average data block length or average record length. The maximum PARM value is '00FFFF' (65,535).

Example: To specify an average data block length of 80 or an average record length of 80, code:

```
KEY    #      LEN    PARM
0009   0001   0003   00 00 50
```

Primary space quantity specification - Key = '000A'

DALPRIME specifies a primary space quantity. You must also code one of the space type keys (DALBLKLN, DALCYL, DALTRK) when you specify DALPRIME. When you code this key, # must be one, LEN must be three, and PARM contains the primary quantity value.

Example: To specify a primary quantity of 20, code:

```
KEY    #      LEN    PARM
000A   0001   0003   00 00 14
```

Secondary space quantity specification - Key = '000B'

DALSECND specifies a secondary space quantity. You must also code one of the space type keys (DALBLKLN, DALCYL, DALTRK) when you specify DALSECND. When you code this key, # must be one, LEN must be three, and PARM contains the secondary quantity value.

Example: To specify a secondary space quantity of 10, code:

```
KEY    #      LEN    PARM
000B   0001   0003   00 00 0A
```

Directory block specification - Key = '000C'

DALDIR specifies the number of blocks to be contained in the directory of a partitioned data set. You may also specify a space type key (DALBLKLN, DALCYL, or DALTRK) and the primary quantity key (DALPRIME) when coding DALDIR. With SMS, the number of blocks that you specify with DALDIR overrides the number that is specified in the data class of the data set. When you code this key, # must be one, LEN must be three, and PARM contains the number of directory blocks.

Example: To specify two directory blocks, code:

```
KEY    #      LEN    PARM
000C   0001   0003   00 00 02
```

Unused space release (RLSE) specification - Key = '000D'

DALRLSE specifies that space allocated to an output data set, but not used, is to be released when the data set is closed. Unused space is released only if the data set is open for output and the last operation was a write. When you code this key, # must be zero and LEN and PARM must not be coded.

Example: To specify the release of unused space, code:

```
KEY    #      LEN   PARM
000D   0000   -     -
```

Format of allocated space specification - Key = '000E'

DALSPFRM specifies a particular format of allocated space. When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'02' Different areas of contiguous space are to be allocated (ALX)

X'04' Maximum contiguous space is required (MXIG)

X'08' Space must be contiguous (CONTIG)

Example: To specify contiguous space format, code:

```
KEY    #      LEN    PARM
000E   0001   0001   08
```

Whole cylinder allocation (ROUND) specification - Key = '000F'

DALROUND specifies that allocated space is to be equal to one or more whole cylinders when requested in units of blocks. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify allocation of whole cylinders, code:

```
KEY    #      LEN   PARM
000F   0000   -     -
```

Volume serial specification - Key = '0010'

DALVLSER specifies volume serial numbers. It is mutually exclusive with the SYSOUT (DALSYSOU) and volume reference (DALVLRDS) keys. When you code DALVLSER, # contains the number of volume serials being specified, LEN contains the length of the immediately following volume serial, and PARM contains the volume serial. Do not specify duplicate volume serial numbers in DALVLSER. Each volume must have a unique volume serial number, regardless of whether it is a tape or disk volume.

Example: To specify the volume serials 231400 and 231401, code:

```
KEY    #      LEN   PARM                 LEN   PARM
0010   0002   0006  F2 F3 F1 F4 F0 F0    0006  F2 F3 F1 F4 F0 F1
```

Private volume specification - Key = '0011'

DALPRIVT specifies that the volume(s) allocated are to be assigned the volume use attribute of private. This key is mutually exclusive with the SYSOUT key (DALSYSOU). When you code DALPRIVT, # must be zero; LEN and PARM are not specified.

Example: To specify the private volume attribute, code:

```
KEY    #      LEN   PARM
0011   0000   -     -
```

Volume sequence number specification - Key = '0012'

DALVLSEQ specifies which volume, of a multi-volume data set, processing is to begin with. This key is mutually exclusive with the SYSOUT key (DALSYSOU). When you code DALVLSEQ, # must be one, LEN must be two, and PARM contains the volume sequence number. The maximum PARM value is '00FF' (255).

Example: To specify a volume sequence number of two, code:

```
KEY    #      LEN    PARM
0012   0001   0002   0002
```

Volume count specification - Key = '0013'

DALVLCNT specifies the maximum number of volumes an output data set may require. This key is mutually exclusive with the SYSOUT key (DALSYSOU). When you code DALVLCNT, # and LEN must be one, and PARM contains the volume count.

Example: To specify a volume count of 10, code:

```
KEY    #      LEN    PARM
0013   0001   0001   0A
```

Volume reference to a dsname specification - Key = '0014'

DALVLRDS indicates that the system is to obtain volume serial information from the specified cataloged data set. This key is mutually exclusive with the SYSOUT (DALSYSOU) and volume serial (DALVLSER) keys. (You cannot use a volume reference to a ddname for dynamic allocation.) For a given job step, the system allows a maximum of 1489 backward references (for a VOL=REF to a dsname or DCB=dsname) when the data set names are 44 characters long.

When you code this key, # must be one, LEN is the actual length of the dsname, and PARM contains the dsname (a name of all blanks is invalid).

Example: To specify volume reference to the data set DSN1, code:

```
KEY    #      LEN    PARM
0014   0001   0004   C4 E2 D5 F1
```

Device description specification - Key = '0015'

DALUNIT specifies a device as a group (esoteric) name, a device type (generic), or a specific device number (in EBCDIC). When you code DALUNIT, # must be one, LEN is the actual length of the device description, and PARM contains the device description. To code a 4-digit specific unit address, precede the EBCDIC device number, PARM, with a slash (/). Include the slash as a character when determining the length. Failure to include the slash will cause the specified 4-digit number to be treated as a device type instead of a device address. A 3-digit specific unit address may also be preceded with a slash.

Example: To specify the group name SYSDA, code:

```
KEY    #      LEN    PARM
0015   0001   0005   E2 E8 E2 C4 C1
```

Example: To specify the device type 3390, code:

```
KEY    #      LEN    PARM
0015   0001   0004   F3 F3 F9 F0
```

Example: To specify the device number 230, code:

```
KEY    #      LEN    PARM
0015   0001   0003   F2 F3 F0
```

Example: To specify the device number 2302, which must be preceded by an EBCDIC slash (61), code:

```
KEY    #      LEN    PARM
0015   0001   0005   61 F2 F3 F0 F2
```

Device count specification - Key = '0016'

DALUNCNT specifies the number of devices to be allocated. It is mutually exclusive with the parallel mount key (DALPARAL). When you code DALUNCNT, # and LEN must be one, and PARM contains the device count. The maximum PARM value is '3B' (59).

Example: To specify a device count of ten, code:

```
KEY    #      LEN    PARM
0016   0001   0001   0A
```

Parallel mount specification - Key = '0017'

DALPARAL specifies that each volume of a data set is to be mounted on a separate device. It is mutually exclusive with the unit count key (DALUNCNT). When you code DALPARAL, # must be zero; LEN and PARM are not specified.

Example: To specify parallel mount, code:

```
KEY    #      LEN   PARM
0017   0000   -     -
```

SYSOUT specification - Key = '0018'

DALSYSOU specifies that a system output data set is to be allocated and defines the output class of the data set. When you code this key and want a class other than the default, # and LEN must be one, and PARM contains the output class. To use the default output class, code zero in the # field; LEN and PARM are not specified. The default output class is determined as follows:

- From the OUTPUT DD statement, if present
- From the default message class, if there is no OUTPUT DD statement.

DALSYSOU is mutually exclusive with the following text unit keys:

- DALSTATS, DALNDISP and DALCDISP
- DALVLSER, DALPRIVT, DALVLSEQ, DALVLCNT, and DALVLRDS
- DALQNAME
- DALSSNM, DALSSPRM, and DALSSATT

Note: Using DALSYSOU with any of these keys will cause dynamic allocation to issue an error code 0380.

Example: To specify a SYSOUT data set in class A, code:

```
KEY   #     LEN   PARM
0018  0001  0001  C1
```

Example: To specify a SYSOUT data set and to default the class, code:

```
KEY   #     LEN  PARM
0018  0000  -    -
```

SYSOUT program name specification - Key = '0019'

DALSPGNM specifies the SYSOUT program name. The SYSOUT key (DALSYSOU) must also be specified when you code DALSPGNM. The subsystem name request (DALSSNM), subsystem parameter (DALSSPRM), and SYSOUT userid (DALUSRID) keys are mutually exclusive with DALSPGNM. When you code this key, # must be one, LEN is the length of the program name, and PARM contains the program name. The maximum length is 8.

Example: To specify the program name MYWRITER, code:

```
KEY   #     LEN   PARM
0019  0001  0008  D4 E8 E6 D9 C9 E3 C5 D9
```

SYSOUT form number specification - Key = '001A'

DALSFMNO specifies the SYSOUT form number. The SYSOUT (DALSYSOU) key must also be specified when you code DALSFMNO. The subsystem name request (DALSSNM) and subsystem parameter (DALSSPRM) keys are mutually exclusive with DALSFMNO. When you code this key, # must be one, LEN is the length of the form number, up to a maximum of 4, and PARM contains the form number.

Example: To specify the form number 1234, code:

```
KEY   #     LEN   PARM
001A  0001  0004  F1 F2 F3 F4
```

SYSOUT output limit specification - Key = '001B'

DALOUTLM specifies the number of logical records in a SYSOUT data set. The SYSOUT key (DALSYSOU) must also be specified when you code DALOUTLM. When you code this key, # must be one, LEN must be three, and PARM contains the output limit.

Example: To specify an output limit of 1000, code:

```
KEY   #     LEN   PARM
001B  0001  0003  00 03 E8
```

Unallocation at CLOSE specification - Key = '001C'

DALCLOSE requests unallocation when a DCB is closed rather than at step unallocation. When you code DALCLOSE, # must be zero; LEN and PARM are not specified. See the description of FREE=CLOSE in *z/OS MVS JCL Reference* for restrictions on the FREE parameter.

Note: The DALCLOSE text unit is ignored for VSAM data sets.

Example:: To specify unallocation at CLOSE, code

```
KEY   #     LEN  PARM
001C  0000  -    -
```

SYSOUT copies specification - Key = '001D'

DALCOPYS requests up to 255 hardcopy listings of a particular SYSOUT data set. The SYSOUT key (DALSYSOU) must also be specified when you code DALCOPYS. When you code this key, # and LEN must be one, and PARM contains the number of copies being requested.

Example:: To specify a request for 25 copies, code

```
KEY   #     LEN   PARM
001D  0001  0001  19
```

Label type specification - Key = '001E'

DALLABEL specifies the type of label associated with a volume. It is mutually exclusive with the SYSOUT (DALSYSOU) key. When you code DALLABEL, # and LEN must be one, and PARM contains one of the following values:

X'01' The volume has no label (NL)

X'02' The volume has an IBM standard label (SL)

X'04' The volume has a non-standard label (NSL)

X'0A' The volume has both an IBM standard label and a user label (SUL)

X'10' Label processing is to be bypassed (BLP)

X'21' The system is to check for and bypass a leading tape mark on DOS unlabeled tape (LTM)

X'40' The volume has an American National Standard label (AL)

X'48' The volume has an American National Standard label and an American National Standard user label (AUL)

Example:: To specify no labels, code

```
KEY   #     LEN   PARM
001E  0001  0001  01
```

Note: If your installation has not specified the BLP feature in the JES2 reader cataloged procedure, specifying BLP has the same effect as specifying NL.

Data set sequence number specification - Key = '001F'

DALDSSEQ specifies the relative position of a data set on a tape volume (data set sequence number). It is mutually exclusive with the SYSOUT (DALSYSOU) key. When you code DALDSSEQ, # must be one, LEN must be two, and PARM contains'' the sequence number. The maximum PARM value is 270F (9999).

Example: To specify a data set sequence number of: 2, code

```
KEY   #     LEN   PARM
001F  0001  0002  00 02
```

Password protection specification - Key = '0020'

DALPASPR specifies that the data set being created is to be password protected. It is mutually exclusive with the SYSOUT (DALSYSOU) key. When you code DALPASPR, # and LEN must be one, and PARM contains one of the following values:

X'10' The data set should not be read, changed, extended, or deleted without the password.

X'30' The data set should not be changed, extended, or deleted without the password. Reading is permitted.

Example: To specify complete password protection,: code

```
KEY   #     LEN   PARM
0020  0001  0001  10
```

Input only or output only specification - Key = '0021'

DALINOUT specifies that the data set is to be processed for input only or output only. In the case of BDAM and BSAM data sets, this key overrides OPEN macro options (INOUT, UPDAT, OUTIN, OUTINX) the same way the JCL LABEL parameter options IN and OUT do. See *z/OS MVS JCL Reference* for details.

DALINOUT is mutually exclusive with the SYSOUT (DALSYSOU) key. When you code DALINOUT, # and LEN must be one, and PARM contains one of the following values:

X'40' Output only is to be requested.

X'80' Input only is to be requested.

Example:: To specify processing for input only, code

```
KEY   #     LEN   PARM
0021  0001  0001  80
```

Expiration date specification (short form) - Key = '0022'

DALEXPDT specifies the date when the data set can be deleted or overwritten by another data set. This key is mutually exclusive with the long form of the expiration date (DALEXPDL), the retention period (DALRETPD), and SYSOUT (DALSYSOU) keys. IBM recommends that you use the DALEXPDL rather than DALEXPDT to specify an expiration date.

When you code DALEXPDT, # must be one, LEN must be five, and PARM contains– five digits a two-digit year value and a three-digit day value (yyddd).

Example: To specify an expiration date of January: 1, 1985 (85001), code

```
KEY   #     LEN   PARM
0022  0001  0005  F8 F5 F0 F0 F1
```

Retention period specification - Key = '0023'

DALRETPD specifies the number of days that must pass before the data set can be deleted or overwritten by another data set. It is mutually exclusive with the expiration date (DALEXPDT) and SYSOUT (DALSYSOU) keys. When you code DALRETPD, # must be one, LEN is the length of the retention period (minimum of 2, maximum of 3), and PARM contains the retention period. The maximum PARM value is X'16B48' or 93000.

Example: To specify a retention period of 10 days, code:

```
KEY   #     LEN   PARM
0023  0001  0002  000A
```

Example: To specify a retention period of 93000 days, code:

```
KEY   #     LEN   PARM
0023  0001  0003  01 6B 48
```

Dummy data set specification - Key = '0024'

DALDUMMY requests that a dummy data set be allocated. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To request allocation of a dummy data set,: code

```
KEY   #     LEN  PARM
0024  0000  -    -
```

Forms control buffer (FCB) image identification specification - Key = '0025'

DALFCBIM specifies the code that identifies the image to be loaded into the forms control buffer (FCB). It is mutually exclusive with the DCB INTVL (DALINTVL) and FRID (DALFRID) keys (described in “DCB attribute text units” on page 666). When you code DALFCBIM, # must be one, LEN contains the length of the image-id (maximum of 4), and PARM contains the image-id.

Example:: To specify the image-id STD1, code

```
KEY   #     LEN   PARM
0025  0001  0004  E2 E3 C4 F1
```

Form alignment and image verification specification - Key = '0026'

DALFCBAV requests that the operator be prompted to check the alignment of the printer forms before the data set is printed, or to visually verify the image displayed on the printer as the desired one. The FCB image-id (DALFCBIM) key must also be coded when DALFCBAV is specified. When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'04' Verification is requested (VERIFY).

X'08' Alignment is requested (ALIGN).

Example:: To specify verification, code

```
KEY   #     LEN   PARM
0026  0001  0001  04
```

QNAME specification - Key = '0027'

DALQNAME specifies the name of a TPROCESS macro. The dsname (DALDSNAM), member name (DALMEMBR), IPLTXTID (DALIPLTX), and SYSOUT (DALSYSOU) keys are mutually exclusive with DALQNAME. The DCB BLKSIZE (DALBLKSZ), BUFL (DALBUFL), LRECL (DALLRECL), OPTCD (DALOPTCD) and RECFM (DALRECFM) keys (see "DCB attribute text units" on page 666) are meaningful with DALQNAME.

When you code this key, # must be one, LEN is the length of the entire process name (maximum of eight characters for each name, plus a period if you are coding two names; total of 17), and PARM contains the process name itself.

Example:: To specify the process name TP1, code

```
KEY   #     LEN   PARM
0027  0001  0003  E3 D7 F1
```

Terminal specification - Key = '0028'

DALTERM specifies that a time-sharing terminal is to be used as an I/O device. In a batch environment, the specification is not used, but is checked for syntax. In a time-sharing environment, all other specifications except DCB specifications are ignored when DALTERM is coded. When you code this key, # must be zero; LEN and PARM are not specified.

Example:: To specify a terminal allocation, code

```
KEY    #      LEN   PARM
0028   0000   -     -
```

Universal character set (UCS) specification - Key = '0029'

DALUCS identifies a special character set to be used for printing a data set. The DCB INTVL (DALINTVL) and RESERVE (DALRSRVF and DALRSRVS) keys (see "DCB attribute text units" on page 666) are mutually exclusive with DALUCS. When you code this key, # must be one, LEN is the length of the character set name code (maximum is four) and PARM contains the character set code.

Example:: To specify the character set code AN, code

```
KEY    #      LEN    PARM
0029   0001   0002   C1 D5
```

Fold mode specification - Key = '002A'

DALUFOLD specifies that the chain or train corresponding to the desired character set is to be loaded in the fold mode. You must also specify the universal character set key (DALUCS) when you code DALUFOLD. When you code this key, # must be zero; LEN and PARM are not specified.

Example:: To specify fold mode, code

```
KEY    #      LEN   PARM
002A   0000   -     -
```

Character set image verification specification - Key = '002B'

DALUVRFY requests that the operator be prompted to verify that the correct chain or train is mounted before the data set is printed. You must also specify the universal character set key (DALUCS) when you code DALUVRFY. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify character set image verification,: code

```
KEY    #      LEN   PARM
002B   0000   -     -
```

DCB reference to a dsname specification - Key = '002C'

DALDCBDS specifies that DCB information is to be retrieved from the data set label of a cataloged data set. This data set must reside on a direct access volume and the volume must currently be mounted.

The DSORG, RECFM, OPTCD, BLKSIZE, LRECL, and KEYLEN DCB attributes, and the volume sequence number and expiration date are copied from the data set label. If text units for those parameters are coded in addition to this key, the text unit specifications override the parameters copied from the data set label.

DALDCBDS is mutually exclusive with DCB reference to a ddname (DALDCBDD).

For a given job, the system allows a maximum of 1489 backward references (for a VOL=REF to a dsname or DCB=dsname) when the data set names are 44 characters long. When DALDCBDS is specified, # must be one, LEN is the length of the dsname, and PARM contains the data set name. (A dsname of all blanks is invalid.)

Example: To specify DCB reference to the dsname ABC,: code

```
KEY    #      LEN    PARM
002C   0001   0003   C1 C2 C3
```

DCB reference to a ddname specification - Key = '002D'

DALDCBDD specifies that DCB information is to be retrieved from the currently allocated data set associated with the specified ddname. For time-sharing users, the expiration date and INPUT/OUTPUT ONLY specifications are also retrieved. This key is mutually exclusive with DCB reference to a dsname (the DALDCBDS key). Any DCB attributes, expiration date (DALEXPD), and INPUT/OUTPUT ONLY (DALINOUT) keys specified in addition to this key override the corresponding DCB parameters associated with the ddname.

When you code DALDCBDD, # must be one, LEN is the length of the ddname, and PARM contains the ddname.

Example: To specify DCB reference to the ddname DD1,: code

```
KEY    #      LEN    PARM
002D   0001   0003   C4 C4 F1
```

SYSOUT remote work station specification - Key = '0058'

In a JES2 environment, DALSUSER requests that, upon deallocation, the SYSOUT data set being allocated be routed to a LOCAL or ANYLOCAL, JES2-defined symbolic name, node, node and work station, remote work station, special local, or user ID.

In a JES3 environment, DALSUSER requests that, upon deallocation, the SYSOUT data set being allocated be routed to ANYLOCAL, JES3-defined device name, device number, or JES3-defined symbolic group name, or NJE node. When coded in conjunction with the user ID key (DALUSRID), this key represents the node to which the user ID, remote workstation, or special local is assigned. This key is equivalent to the JCL **DEST=** parameter, where **DEST=(***dalsuser***)** or **DEST=(***dalsuser,dalusrid***)**. The SYSOUT key (DALSYSOU) is required with this key.

When you code DALSUSER, # must be one, LEN is the length of the destination specified in PARM, up to a maximum of 8, and PARM contains the work station name, node, special local, or user ID.

Example:: To specify the work station USER01, code

```
KEY    #      LEN    PARM
0058   0001   0006   E4 E2 C5 D9 F0 F1
```

SYSOUT hold queue specification - Key = '0059'

DALSHOLD requests that the SYSOUT data set being allocated be placed on the hold queue when it is deallocated. The SYSOUT key (DALSYSOU) must also be specified when DALSHOLD is specified.

Note: If the data set being allocated is the internal reader, then specifying DALSHOLD will cause the job to be held after conversion and before execution.

When you code this key, # must be zero; LEN and PARM are not specified.

Example:: To specify hold, code

```
KEY    #      LEN   PARM
0059   0000   -     -
```

Subsystem name request specification - Key = '005F'

DALSSNM specifies a subsystem name. You must specify the name of the subsystem that is to process the request for allocation unless you want the request processed by the default subsystem.

- When you code DALSSNM to request a subsystem other than the default subsystem, # must be one, LEN specifies the length of the subsystem name (maximum of four) and PARM contains the subsystem name (one to four characters).

  The first character of the subsystem name must be either alphabetic or national and the remaining characters must be either alphameric or national. See *z/OS MVS JCL Reference* for a list of the alphameric and national character sets.
- When you code DALSSNM to request the default subsystem, # must be zero; LEN and PARM are not be specified.

DALSSNM is mutually exclusive with the SYSOUT (DALSYSOU), SYSOUT program name (DALSPGNM) and SYSOUT form number (DALSFMNO) keys.

Your installation's system programming staff can identify the subsystems at your installation that support DALSSNM requests.

Example 1:: To request subsystem SUB1, code

```
KEY    #      LEN    PARM
005F   0001   0004   E2 E4 C2 F1
```

Example 2:: To request the default subsystem, code

```
KEY    #      LEN   PARM
005F   0000   -     -
```

Subsystem parameter specification - Key = '0060'

DALSSPRM specifies parameters that will be processed by a subsystem. When coding DALSSPRM, you must also specify the subsystem name (DALSSNM) key. DALSSPRM is mutually exclusive with the SYSOUT (DALSYSOU), SYSOUT program name (DALSPGNM), and SYSOUT form number (DALSFMNO) keys.

When you code this key, # contains the number of parameters to be passed to the subsystem (maximum of 254), LEN specifies the length of the immediately-following parameter (value range from 0 to 67), and PARM contains the parameter to be passed to the subsystem. When you code a LEN value of 0, do not code a PARM value.

Example: To specify two parameters, PARM1 and PARAMETER2,: code

```
KEY    #      LEN    PARM             LEN    PARM
0060   0002   0005   D7 C1 D9 D4 F1   000A   D7 C1 D9 C1 D4 C5 E3 C5 D9 F2
```

Note: For additional information about subsystem data sets and subsystem parameters, refer to the documentation for the particular subsystem.

PROTECT specification - Key = '0061'

DALPROT requests that the specified direct access data set or tape volume be RACF-protected when defined (DASD) or used (tape). It is mutually exclusive with the SYSOUT (DALSYSOU), FCB (DALFCBIM), QNAME (DALQNAME), terminal (DALTERM) and UCS (DALUCS) keys.

When you code DALPROT, # must be zero; LEN and PARM are not specified. See *z/OS MVS JCL Reference* for additional information about specifying the PROTECT function.

Example:: To specify PROTECT, code

```
KEY    #      LEN   PARM
0061   0000   -     -
```

SYSOUT user ID specification - Key = '0063'

In a JES2 environment, DALUSRID requests that, upon deallocation, the SYSOUT data set being allocated be routed to the specified user ID, JES2 remote device, or JES2 special local device.

In a JES3 environment, DALUSRID requests that, upon deallocation, the SYSOUT data set being allocated be routed to the specified user ID, or JES3-defined device name.

The SYSOUT (DALSYSOU) and SYSOUT remote work station (DALSUSER) keys are required with this key. This key is equivalent to the JCL **DEST=** parameter, where **DEST=(***dalsuser,dalusrid***)**. The SYSOUT program name key (DALSPGNM) is mutually exclusive with DALUSRID. If these keys are used together, dynamic allocation will issue an error code 0380.

When you code this key, # must be one, LEN is the length of the destination specified in PARM, up to a maximum of 8, and PARM contains the user ID, remote device, or special local device. The user ID may be any EBCDIC characters, including special characters.

Example: To send the Class A SYSOUT data set to user: ID D58-VWM at remote work station (node) DALLAS, code

```
KEY    #      LEN    PARM
0063   0001   0007   C4 F5 F8 60 E5 E6 D4
0018   0001   0001   C1
0058   0001   0006   C4 C1 D3 D3 C1 E2
```

Burst specification - Key = '0064'

DALBURST specifies which stacker of the 3800 Printing Subsystem is to receive the paper output.

When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'02' Burster-trimmer-stacker

X'04' Continuous form stacking

Example:: To specify continuous form stacking, code

```
KEY    #      LEN    PARM
0064   0001   0001   04
```

Character arrangement table specification - Key = '0065'

DALCHARS specifies the name or names of character arrangement tables for printing a data set on the 3800 Printing Subsystem.

When you code this key, # contains the number of character arrangement tables being specified up to a maximum of 4. LEN contains the length of the immediately-following character arrangement table, up to a maximum of 4, and PARM contains the name of the character arrangement table.

Example: To specify the character arrangement tables: GS10 and GS12, code

```
KEY    #      LEN    PARM       LEN    PARM
0065   0002   0004   C7E2F1F0   0004   C7E2F1F2
```

Copy groups specification - Key = '0066'

DALCOPYG specifies how multiple copies of 3800 output are to be grouped. The copies specification (DALCOPYS) key is required with this key.

When you code DALCOPYG, # contains the number of group values being specified (up to a maximum of 8), LEN must be one, and PARM contains the number of copies of each page that are to be grouped together.

Example: To indicate that six copies of the data set are to be printed in three groups; and that the first group is to contain one copy of each page, the second group is to contain three copies of each: page, and the third group is to contain two copies of each page, code

```
KEY    #      LEN    PARM   LEN    PARM   LEN    PARM
001D   0001   0001   06
0066   0003   0001   01     0001   03     0001   02
```

Flash forms overlay specification - Key = '0067'

DALFFORM specifies the forms overlay to be used on the 3800 Printing Subsystem.

When you code this key, # must be one, LEN contains the length of the form name, up to a maximum of 4. and PARM contains the name of the forms overlay frame that the operator is to insert into the printer before printing begins.

Example: To specify the forms overlay frame named: ABCD, code

```
KEY    #      LEN    PARM
0067   0001   0004   C1C2C3C4
```

Flash forms overlay count specification - Key = '0068'

DALFCNT specifies the number of copies on which the forms overlay is to be printed. When specifying DALFCNT, you must also specify the flash forms overlay (DALFFORM) key.

When you code DALFCNT, # and LEN must be one, and PARM contains the number of copies.

Example: To specify that the first five copies are: to be flashed with the forms overlay, code

```
KEY    #      LEN    PARM
0068   0001   0001   05
```

Copy modification module specification - Key = '0069'

DALMMOD specifies the name of the copy modification module to be loaded into the 3800 Printing Subsystem.

When you code DALMMOD, # must be one, LEN contains the length of the module name up to a maximum of 4, and PARM contains the name of the copy modification module.

Example: To specify that the data in the copy modification: module named A is to replace the variable data in the data set, code

```
KEY    #      LEN    PARM
0069   0001   0001   C1
```

Copy module table reference specification - Key = '006A'

DALMTRC specifies the table reference character that corresponds to a character arrangement table specified on the DALCHARS text unit key, and used for printing the copy modification data. When specifying DALMTRC, you must also specify the copy modification module specification (DALMMOD) key.

When you code this key, # and LEN must be one and PARM contains one of the following values:

X'00' The first character arrangement table specified on the DALCHARS text unit

X'01' The second character arrangement table specified

X'02' The third character arrangement table specified

X'03' The fourth character arrangement table specified

Example: To indicate that the first character arrangement: table specified on the DALCHARS key is to be used, code

```
KEY    #      LEN    PARM
006A   0001   0001   00
```

DEFER specification - Key = '006C'

DALDEFER specifies that the system should allocate a device to the data set, but the volume(s) on which the data set resides should not be mounted until the data set is opened.

When you code DALDEFER, # must be zero; LEN and PARM are not specified. See *z/OS MVS JCL Reference* for the rules regarding the use of DEFER.

Example: To specify a request for deferred mounting: of a volume or volumes, code

```
KEY    #      LEN   PARM
006C   0000   -     -
```

EXPIRATION DATE specification (long form) - Key = '006D'

DALEXPDL specifies the date when the data set can be deleted or overwritten by another data set. The key is mutually exclusive with the retention period (DALRETPD), SYSOUT (DALSYSOU), and expiration date short form (DALEXPDT) keys. IBM recommends that you use DALEXPDL rather than DALEXPDT to specify an expiration date. When you code DALEXPDL, # must be 1, LEN must be 7, and PARM must contain— seven digits a four-digit year value and a three-digit day value (yyyyddd).

Example: To specify an expiration date of January: 1, 2005 (2005001), code

```
KEY    #      LEN   PARM
006D   0001   0007  F2 F0 F0 F5 F0 F0 F1
```

Override job affinity - Key = '0070'

DALOVAFF applies to internal readers only. It specifies that any job submitted to the internal reader must run on the system on which the job is submitted. DALOVAFF overrides any affinity specification within the JCL or default affinity for the input device or job class. Only an authorized program may use this text unit.

When you specify DALOVAFF, you must code:

- DALSYSOU (SYSOUT)
- DALSPGNM (SYSOUT program name) with a parameter length of 6 and a parameter value of INTRDR

This key is mutually exclusive with the following text units: DALSTATS, DALNDISP, DALCDISP, DALVLSER, DALPRIVT, DALVLSEQ, DALVLCNT, DALVLRDS, DALLABEL, DALDSSEQ, DALPASPR, DALINOUT, DALEXPDT, DALRETPD, DALQNAME, DALSSNM, DALSSPRM, DALPROT, DALEXPDL, DALSSATT, and DALUSRID.

When you code DALOVAFF, # must be 0; LEN and PARM are not specified.

Example: To specify that a job submitted to the internal reader must run on the system on which the job is submitted, code:

```
KEY    #      LEN   PARM
0070   0000    -     -
```

CTOKEN return specification - Key = '0071'

DALRTCTK requests that the JES Client Token (CTOKEN) associated with the SYSOUT allocation be returned to the caller of DYNALLOC. When you code DALRTCTK, # must be 1, and LEN must be hex '0050' (decimal 80) and PARM is an 80-byte field.

Dynamic allocation places the allocated CTOKEN in PARM upon completion of the associated SYSOUT allocation request.

When you specify DALRTCTK, you must also specify the SYSOUT key (DALSYSOU - X'0018').

Example: To request that the CTOKEN associated with: the allocated SYSOUT dataset be returned, code

```
KEY    #      LEN   PARM
0071   0001   0050  --------- ... -------- 80 bytes in length
```

This specification would be updated upon the assignment of the CTOKEN as: follows

```
KEY    #      LEN   PARM
0071   0001   0050  xxxxxxxxx ... xxx JES-supplied 80-byte CTOKEN
```

SMSHONOR specification – Key = '0076'

DALSMSHR indicates that the system must honor the device name or the esoteric specified on the unit name for an SMS tape library request. When you code DALSMSHR, # must be zero; LEN and PARM are not specified. See *z/OS MVS JCL Reference* for the rules regarding the use of SMSHONOR.

Example: To specify a request to honor the unit name for an SMS tape library request, code

```
KEY    #      LEN   PARM
0076  0000     -     -
```

Access specification - Key = '8001'

DALACODE specifies an accessibility code for protecting ISO/ANSI/FIPS Version 3 and ISO/ANSI Version 4 labelled tape data sets.

When you code DALACODE, # must be 1, LEN is the length of the accessibility code (maximum of 8), and PARM contains the accessibility code. For ISO/ANSI/FIPS Version 3, the accessibility code must begin with an uppercase letter from– A Z. For ISO/ANSI Version 4, the accessibility code must begin with— an uppercase letter from A Z, a number from 0 9, or one of: the following special characters

```
%&:<! * "   ' ( ) + , - . /  ;  = > ? _
```

See *z/OS MVS JCL Reference* for more information about the DD ACCODE parameter.

Example: To specify an accessibility code of Z for: an ANSI tape data set, code

```
KEY    #      LEN   PARM
8001   0001   0001  E9
```

OUTPUT statement reference - Key = '8002'

DALOUTPT explicitly associates a SYSOUT data set with the OUTPUT JCL statement specified in the PARM field or an output descriptor created by OUTADD. See Chapter 27, “Dynamic output,” on page 719 for a description of OUTADD. The SYSOUT (DALSYSOU) text unit key is required with this key.

When you code DALOUTPT, # contains the number of output statements being specified, up to a maximum of 128 (X'0080'), LEN is the length of the OUTPUT statement, up to a maximum of 26 (X'001A'), and PARM contains the name of the OUTPUT statement in one of the following forms:

- name
- stepname.name

- stepname.procstepname.name

Example: To reference an OUTPUT JCL statement named OUT1 in the job step named STEP1, and another named OUTX in the current step,: code

```
KEY    #      LEN   PARM                           LEN   PARM
8002   0002   000A  E2 E3 C5 D7 F1 4B D6 E4 E3 F1  0004  D6 E4 E3 E7
```

CNTL specification - Key = '8003'

DALCNTL references a JCL CNTL statement that appears earlier in the job. The system searches for an earlier CNTL statement with a label that matches the label specified in the PARM field.

Use DALCNTL to dynamically allocate a printer in direct-printing mode. For more information, see *z/OS Communications Server: IP Network Print Facility*

When you code DALCNTL, # must be one, LEN is the length of the CNTL statement label, up to a maximum of 26 (X'001A'), and PARM contains the name of the CNTL statement label in one of the following forms:

- name
- stepname.name
- stepname.procstepname.name

Example: To request the system to use the program"" control statements following the CNTL statement named WKLYPGM located: in this step or preceding the first step, code

```
KEY    #      LEN   PARM
8003   0001   0007  E6 D2 D3 E8 D7 C7 D4
```

Storage class specification - Key = '8004'

DALSTCL specifies the storage class of an SMS-managed data set. When you code DALSTCL, # must be one, LEN is the length of the storage class name, up to a maximum of 8, and PARM contains the name of the storage class.

Example:"" To specify the storage class of SAM for: an SMS-managed data set, code

```
KEY    #      LEN    PARM
8004   0001   0003   E2 C1 D4
```

Management class specification - Key = '8005'

DALMGCL specifies the management class of an SMS-managed data set. When you code DALMGCL, # must be one, LEN is the length of the management class name (up to 8 characters), and PARM contains the name of the management class.

Example:"" To specify the management class, SAM,: of an SMS-managed data set, code

```
KEY    #      LEN    PARM
8005   0001   0003   E2 C1 D4
```

Data class specification - Key = '8006'

DALDACL specifies the data class of the data set. When you code DALDACL, # must be one, LEN is the length of the data class name (up to 8 characters), and PARM contains the name of the data class.

Example:“” To specify the data class of SAM for: an SMS-managed data set, code

```
KEY    #      LEN    PARM
8006   0001   0003   E2 C1 D4
```

Record organization specification - Key = '800B'

DALRECO specifies the record organization of a VSAM data set. When you code DALRECO, # and LEN must be 1, and PARM contains one of the following:

X'80' VSAM key-sequenced data set

X'40' VSAM entry-sequenced data set

X'20' VSAM relative record data set

X'10' VSAM linear space data set

Example: To specify a key-sequenced record organization,: code

```
KEY    #      LEN    PARM
800B   0001   0001   80
```

Key offset specification - Key = '800C'

DALKEYO specifies the key offset. The key offset is the position of the first byte of the key in each logical record of a the specified VSAM data set. If the key is at the beginning of the logical record, the offset is zero. When you code DALKEYO, # must be 1 and LEN must be 4, and PARM contains the key offset.

Example: To specify a key offset of 18 decimal (12: hexadecimal) bytes, code

```
KEY    #      LEN    PARM
800C   0001   0004   12
```

Copy DD specification - Key = '800D'

DALREFD specifies the name of the JCL DD statement from which the attributes are to be copied.

The name can be a *ddname*, a *stepname.ddname*, or a *stepname.procstepname.ddname* where *ddname* is the label on a JCL DD statement, and *stepname* and *procstepname* are labels that appear on JCL EXEC statements. When you code DALREFD, # must be 1, and LEN is the length of the name field, up to a maximum of 44 (X'2C'), and PARM contains the name.

Example: To copy the data set attributes from the“”: JCL DD statement named SAM, code

```
KEY    #      LEN    PARM
800D   0001   0003   E2 C1 D4
```

Copy profile specification - Key = '800E'

DALSECM specifies the name of the RACF profile to be copied.

To specify a RACF profile that was defined generically, (such as one defined with the GENERIC subparameter on the JCL DD statement SECMODEL parameter), # must be 2, LEN1 is the length of the profile name, and PARM1 contains the profile name. LEN2 must be 1, and PARM2 must contain X'80'.

To specify a profile that was not defined generically, # must be 1, LEN is the length of the profile name, and PARM contains the profile name. Do not specify additional LEN or PARM fields.

Example:“” To copy the generic RACF profile, RPROF,: code

```
KEY    #      LEN1   PARM1                LEN2   PARM2
800E   0002   0005   D9 D7 D9 D6 C6       0001   80
```

Copy model specification - Key = '800F'

DALLIKE specifies the name of the model data set from which the attributes are to be copied. When you code DALLIKE, # must be one, LEN is the length of the model data set name, up to a maximum of 44 (X'2C'), and PARM contains the data set name.

Example: To copy the attributes of the model data“”: set, SAM, code

```
KEY    #      LEN    PARM
800F   0001   0003   E2 C1 D4
```

Average record specification - Key = '8010'

DALAVGR specifies the allocation unit to be used when the data set is allocated.

When you code DALAVGR, # and LEN must be 1, and PARM is one of the following:

X'80' Represents single-record units.

X'40' Represents thousand-record units.

X'20' Represents million-record units

Example:: To specify single-record units, code

```
KEY    #      LEN    PARM
8010   0001   0001   80
```

Data set type specification - Key = '8012'

DALDSNT specifies the type attribute of a data set as PDS, PDSE, HFS data set, OpenMVS first-in-first-out (FIFO) special file, extended format, basic, or large.

When you code DALDSNT, # and LEN must be 1; PARM must contain one of the following:

X'80' Represents a library (PDSE).

X'40' Represents a partitioned data set (PDS).

X'20' Represents a FIFO special file (PIPE).

X'10' Represents an HFS data set (HFS). This type of UNIX file system is different from a z/OS File System (zFS) that resides in a linear data set.

X'08' Represents an extended format required data set (EXTREQ).

X'04' Represents an extended format preferred data set (EXTPREF).

X'02' Represents a basic format data set (BASIC).

X'01' Represents a large format data set (LARGE = greater than 65,535 tracks).

Example:: To specify a PDS data set, code

```
KEY    #      LEN   PARM
8012   0001   0001  40
```

Spin data set specification - Key = '8013'

DALSPIN specifies whether the output for the SYSOUT data set is to be printed immediately upon unallocation of the data set, or at the end of the job.

When you code DALSPIN, # and LEN must be 1; PARM must contain one of the following:

X'80' Data set available for printing when it is unallocated

X'40' Data set available for printing at the end of the job.

Example: To specify that the output be printed at the end of the job, code:

```
KEY    #     LEN   PARM
8013   0001  0001  40
```

Segment spin data set specification - Key = '8014'

DALSEGM specifies the number of logical, line-mode pages (the segment) to be produced for a SYSOUT data set before the segment becomes eligible for immediate printing.

When you code DALSEGM, # must be 1 and LEN must be 4; PARM must contain a positive hexadecimal integer from X'00000001' to X'0001869F' (equivalent decimal value from 1 to 99,999).

Example: To specify a segment of 100 pages, code:

```
KEY    #     LEN   PARM
8014   0001  0004  00000064
```

z/OS UNIX file path specification - Key = '8017'

DALPATH specifies the pathname of the z/OS UNIX file to be allocated.

You can code DALPATH only with the DALDUMMY, DALTERM, DALDDNAM, DALPOPT, DALPMDE, DALPNDS, DALPCDS, DALBLKSZ, DALBUFNO, DALLRECL, DALNCP, DALRECFM, DALLRECK, DALFDAT and DALDSNT keys. DALPATH is mutually exclusive with all other keys. In JCL, the information conveyed by this key is specified using the PATH keyword.

Dynamic allocation does not support backward or forward references. See Table 69 on page 527.

When you code this key, # must be 1, LEN is the length of the pathname, and PARM contains the pathname. The maximum length of the pathname is 255 characters. Refer to Table 83 on page 656 for values.

If

- You specify either:
  - – The hex integers representing OCREAT alone

    or
  - – The hex integers representing both OCREAT and OEXCL

  on the DALPOPT key,

  And if
- The file does not exist,

Then MVS performs an **open()** function. The options from DALPOPT, the pathname from the DALPATH key, and the options DALPMDE (if specified) are used in the **open()**. MVS uses the **close()** function to close the file before the application program receives control.

For status group options other than OCREAT and OEXCL, the description in this information assumes that the application passes the values to the **open()** function without modification. That is, this application uses dynamic allocation information retrieval (the DYNALLOC macro) to retrieve the values specified for DALPOPT and passes the values to the **open()** function. The application program can ignore or modify the information specified in the JCL.

Example: To specify the z/OS UNIX file pathname **/u/myuid/myapp/scr.dat**:, code

```
KEY    #     LEN   PARM
8017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
```

Example: To specify a dummy z/OS UNIX file, **/dev/null**:, code

```
KEY    #     LEN   PARM
8017   0001  0009  61 84 85 A5 61 95 A4 93 93
```

Note: Specifying the DALDUMMY key with any pathname achieves the same result.

z/OS UNIX file options - Key = '8018'

DALPOPT specifies the file options for the z/OS UNIX file. You can code DALPOPT only when you also code the pathname (DALPATH) key.

If

- You specify either
  - The hex integers representing OCREAT alone

    or
  - The hex integers representing both OCREAT and OEXCL

  on the DALPOPT key,

  And if
- The file does not exist,

Then MVS performs an **open()** function. The options from DALPOPT, the pathname from the DALPATH key, and the options DALPMDE (if specified) are used in the **open()**. MVS uses the **close()** function to close the file before the application program receives control.

For status group options other than OCREAT and OEXCL, the description in this information assumes that the application passes the values to the **open()** function without modification. That is, this application uses dynamic allocation information retrieval (the DYNALLOC macro) to retrieve the value specified for DALPOPT and passes the value to the **open()** function. The application program can ignore or modify the information specified in the JCL.

When you code DALPOPT, # must be 1 and LEN must be 4; PARM must contain one of the following values or an “inclusive OR” result from any combination of the following values:

Table 83. DALPOPT Options

Four-byte Hex Integers	JCL Equivalent
X'00000100'	OSYNC
X'00000080'	OCREAT
X'00000040'	OEXCL
X'00000020'	ONOCTTY
X'00000010'	OTRUNC
X'00000008'	OAPPEND
X'00000004'	ONONBLOCK
X'00000003'	ORDWR
X'00000002'	ORDONLY
X'00000001'	OWRONLY

Example: To specify a file option of read-only for a z/OS UNIX file option specification, code

```
KEY    #     LEN   PARM
8018   0001  0004  00000001
```

Example: To specify a file option of create, if the file does not already exist with the intent to read and write the file, use the “inclusive OR”+: result of OCREATOEXCL and code

```
KEY    #     LEN   PARM
8018   0001  0004  000000C3
```

z/OS UNIX file access attributes - Key = '8019'

DALPMDE specifies the file access attributes for the z/OS UNIX file. You can code DALPMDE only when you also code the pathname (DALPATH) key.

If

- You specify either
  - The hex integers representing OCREAT alone

    or
  - The hex integers representing both OCREAT and OEXCL

  on the DALPOPT key,

  And if
- The file does not exist,

Then MVS performs an **open()** function. The options from DALPOPT, the pathname from the DALPATH key, and the options DALPMDE (if specified) are used in the **open()**. MVS uses the **close()** function to close the file before the application program receives control.

For status group options other than OCREAT and OEXCL, the description in this information assumes that the application passes the values to the **open()** function without modification. That is, this application uses dynamic allocation information retrieval (the DYNALLOC macro) to retrieve the value specified for PATHOPTS and passes the value to the **open()** function. The application program can ignore or modify the information specified in the JCL.

When you code DALPMDE, # must be 1 and LEN must be 4; PARM must contain one of the following values or an “inclusive OR” result from any combination of the following values:

Table 84. DALPMDE Attributes

Four-byte Hex Integers	JCL Equivalent
X'00000800'	SISUID
X'00000400'	SISGID
X'00000100'	SIRUSR
X'00000080'	SIWUSR
X'00000040'	SIXUSR
X'000001C0'	SIRWXU
X'00000020'	SIRGRP
X'00000010'	SIWGRP
X'00000008'	SIXGRP
X'00000038'	SIRWXG
X'00000004'	SIROTH
X'00000002'	SIWOTH

Table 84. DALPMDE Attributes (continued)

Four-byte Hex Integers	JCL Equivalent
X'00000001'	SIXOTH
X'00000007'	SIRWXO

Example: To specify a file access attribute allowing users in the owners access group read access to the z/OS UNIX file being dynamically allocated, code:

```
KEY    #     LEN   PARM
8019   0001  0004  00000020
```

Example: To specify a file access attribute allowing the owner of the file to write the file and users in the same file group class to read the file, use the “inclusive OR”+ expression for SIWUSRSIRGRP: and code

```
KEY    #     LEN   PARM
8019   0001  0004  000000A0
```

z/OS UNIX file normal disposition specification - Key = '801A'

DALPNDS specifies the normal z/OS UNIX file disposition desired. You can code DALPNDS only when you also code the pathname (DALPATH) key.

When you code DALPNDS, # and LEN must be 1, and PARM contains one of the following values:

- X'04' - the file is to be deleted
- X'08' - the file is to be kept

Example: To specify a normal disposition of file: deletion, code

```
KEY    #     LEN   PARM
801A   0001  0001  04
```

z/OS UNIX file abnormal disposition specification - Key = '801B'

DALPCDS specifies the abnormal z/OS UNIX file disposition desired. You can code DALPCDS only when you also code the pathname (DALPATH) key.

When you code DALPCDS, # and LEN must be 1, PARM contains one of the following values:

- X'04' - the file is to be deleted
- X'08' - the file is to be kept

Example: To specify an abnormal disposition of file: deletion, code

```
KEY    #     LEN   PARM
801B   0001  0001  04
```

Record-level sharing specification - Key = '801C'

DALRLS specifies the VSAM record-level sharing (RLS) protocol to be used with a VSAM data set. For more information about using RLS, see *z/OS DFSMS Using Data Sets*.

When you code DALRLS, # and LEN must be 1, and PARM contains one of the following:

X'80' NRI (DALNRI) No read integrity

X'40' CR (DALCR) Consistent read

X'20' CRE (DALCR) Consistent read explicit

Example: The following code specifies that an application program can ensure that records read by a unit of recovery are not changed by other units of recovery until the reading unit of recovery issues a syncpoint:

```
KEY    #     LEN   PARM
801C   0001  0001  20
```

Organization of a z/OS UNIX file - Key = '801D'

DALFDAT specifies the organization of a z/OS UNIX file. You can code DALFDAT only when you also code the pathname (DALPATH) key.

When you code DALFDAT, # and LEN must be 1, and PARM contains one of the following values:

- X'80'— - the file organization is binary records not delimited
- X'40'— - the file organization is text records are EBCDIC delimited by newline characters (x'15')
- X'20'— - the file organization is data records with a prefix indicating the length of the following record.

  Note:
  The record prefix for FILEDATA=RECORD is mapped by the IGGRPFX macro. This is different from the record descriptor word (RDW) that is in z/OS physical sequential format-V data sets.

Example: To specify that a file has records delimited by newline characters, code:

```
KEY    #     LEN   PARM
801D   0001  0001  40
```

VSAM RLS log stream specification - Key = '801F'

DALLGST specifies the prefix of the name of the log stream for the SMS managed VSAM data set.

Example: To specify a Log Stream prefix of 'SSAB1234.NEW', code:

```
KEY    #     LEN   PARM
801F   0001  000C  E2 E2 C1 C2 F1 F2 F3 F4 4B D5 C5 E6
```

CCSID specification - Key = '8020'

DALDCCS specifies the Coded Character Set Identifier indicating the character code conversion performed on reads from and writes to tapes accessed in ISO/ANSI Version 4 format.

When you code DALDCCS, # must be one, LEN must be four, and PARM contains a positive hexadecimal value from X'00000001' to X'0000FFFF' (equivalent decimal value from 1 to 65535). See *z/OS MVS JCL Reference* for more information about the DD CCSID parameter.

```
KEY    #     LEN   PARM
8020   0001  0004  0000FFFF
```

Block size limit specification - Key = '8022'

DALBSLM specifies the maximum length system-determined block size processing is to use when determining a block size appropriate for the media type to which the data is being written. When you code DALBSLM, # must be one, LEN must be 1 to 10, and PARM contains the block size limit in EBCDIC. The maximum PARM value is 2,147,483,648. The minimum value is 32,760.

Example: To specify the maximum block size limit of 2,147,483,648, code:

```
KEY   #     LEN   PARM
8022  0001  000A  F2 F1 F4 F7 F4 F8 F3 F6 F4 F8
```

or (2097152K)

```
KEY   #     LEN   PARM
8022  0001  0008  F2 F0 F9 F7 F1 F5 F2 D2
```

or (2048M)

```
KEY   #     LEN   PARM
8022  0001  0005  F2 F0 F4 F8 D4
```

or (2G)

```
KEY   #     LEN   PARM
8022  0001  0002  F2 C7
```

Example: To specify a block size limit of 20 megabytes, code:

```
KEY   #     LEN   PARM
8022  0001  0003  F2 F0 D4
```

Key label 1 specification - Key = '8023'

DALKYL1 specifies the label for the key encrypting key used by the key manager. The key encrypting key is used to encrypt the data (encryption) key.

When you code DALKYL1, # must be one, LEN must be 1 to 64, and PARM contains the key label in EBCDIC. You must also include the DALKYC1 text unit.

Code the DALKYL1 text unit to do either of the following tasks:

- Specify the label for the key encrypting key used by the encryption key manager.
- Override the label for the key encrypting key defined in the data class of the data set.

Specification of the key labels does not by itself enable encryption. Encryption must be enabled by a data class that specifies an encryption format, for example EEFMT2. For complete documentation on using tape encryption, see *z/OS DFSMS Software Support for IBM System Storage TS1140, TS1130, and TS1120 Tape Drives (3592)*.

Example: To specify a key label 1 of "LABELQ1.LABELQ2.LABELQ3", code:

```
KEY    #     LEN   PARM
8023   0001  0017  D3C1C2C5D3D8F14BD3C1C2C5D3D8F24BD3C1C2C5D3D8F3
```

Key label 2 specification - Key = '8024'

DALKYL2 specifies the label for the key encrypting key used by the key manager. The key encrypting key is used to encrypt the data (encryption) key.

When you code DALKYL2, # must be one, LEN must be 1 to 64, and PARM contains the key label in EBCDIC. You must also include the DALKYC2 text unit.

Code the DALKYL2 text unit to do either of the following tasks:

- Specify the label for the key encrypting key used by the encryption key manager.
- Override the label for the key encrypting key defined in the data class of the data set.

Specification of the key labels does not by itself enable encryption. Encryption must be enabled by a data class that specifies an encryption format, for example EEFMT2. For complete documentation on using tape encryption, see *z/OS DFSMS Software Support for IBM System Storage TS1140, TS1130, and TS1120 Tape Drives (3592)*.

Example: To specify a key label 2 of "LABELQ1.LABELQ2.LABELQ3", code:

```
KEY    #     LEN   PARM
8024   0001  0017  D3C1C2C5D3D8F14BD3C1C2C5D3D8F24BD3C1C2C5D3D8F3
```

Key encode 1 specification - Key = '8025'

DALKYC1 specifies the encoding of the label for the key encrypting key used by the encryption key manager. The key encrypting key is used to encrypt the data (encryption) key.

When you code DALKYC1, # must be 1, LEN must be 1, and PARM contains one of the following values:

L(X'03')
Indicates that the key label 1 is to be stored as part of the EEDK structure on the tape cartridge.

H(X'08')
Indicates that a hash of the public key referenced by key label 1 is to be stored on the cartridge rather than the key label.

You must also include the DALKYL1 text unit.

Code the DALKYC1 text unit when you want to specify the encoding for the label for the key encrypting key used by the encryption key manager. For complete documentation on using tape encryption, see *z/OS DFSMS Software Support for IBM System Storage TS1140, TS1130, and TS1120 Tape Drives (3592)*.

Example: To specify a key encode 1 of "L", code:

```
KEY    #     LEN   PARM
8025   0001  0001  D3
```

Key encode 2 specification - Key = '8026'

DALKYC2 specifies the encoding of the label for the key encrypting key used by the encryption key manager. The key encrypting key is used to encrypt the data (encryption) key.

When you code DALKYC2, # must be 1, LEN must be 1, and PARM contains one of the following values:

X'01' Indicates that No Extended Attributes allowed.

X'02' Indicates that a hash of the public key referenced by key label 2 is to be stored on the cartridge rather than the key label.

You must also include the DALKYL2 text unit.

Code the DALKYC2 text unit when you want to specify the encoding for the label for the key encrypting key used by the encryption key manager. For complete documentation on using tape encryption, see *z/OS DFSMS Software Support for IBM System Storage TS1140, TS1130, and TS1120 Tape Drives (3592)*.

Example: To specify a key encode 2 of "H", code:

```
KEY    #     LEN   PARM
8026   0001  0001  C8
```

Extended attributes specification - Key = '8028'

DALEATT indicates whether the data set can support extended attributes (format 8 and 9 DSCBs). By definition, a data set with extended attributes can reside in EAS (extended address space) on an EAV (extended address volume). These attributes can be specified for non-VSAM data sets as well as for VSAM data sets.

When you code DALEATT, # must be one, LEN must be one, and PARM contains one of the following values:

X'01' No Extended Attributes allowed.

X'02' Extended attributes are optional, and will be created if the data set is created on an EAV. In addition, the data set can be created in the EAS of the EAV.

Example: To specify extended attributes of OPT, code:

```
KEY    #     LEN   PARM
8028   0001  0001  02
```

FREEVOL specification - Key = '8029'

FREEVOL specifies whether to allow other jobs to read freed volumes of a multivolume tape file as the volume is dismounted by the job.

When you code FREEVOL, # must be one, LEN must be one, and PARM contains one of the following values:

X'01' Requests that volumes be dequeued at the end of the job step

X'02' Requests that when reading a multivolume data set, the system finish reading the current volume and then dequeue the volume serial number and demount the volume. This makes the volume immediately available to another job in another system. An attempt by the same task to reprocess the volume using the same JCL DD statement will result in an abnormal end.

Example: To specify that the data set volume is to be freed as it is dismounted, code:

```
KEY    #     LEN   PARM
8029   0001  0001  02
```

SPIN interval for the allocated SYSOUT data set - Key = '802A'

DALSPI2 specifies the interval for spinning a SYSOUT data set when DALSPIN is also coded with a PARM of X'80''.

When you code DALSPI2, # must be one, LEN is 3 to 8 and the PARM field is the character (EBCDIC) representation of the SPIN interval, such as F9F9F9 for '999' or F1F27AF0F0 for '12:00' with the following values:

hh:mm
Data set will be spun at the time **‘hh:mm'** each 24 hour period where **‘hh'** is hours and has a range of 00 through 23 and **‘mm'** is minutes and has a range of 00 through 59. The time must be specified within apostrophes.

+hh:mm
Data set will be spun every **'hh:mm'** time interval where **‘hh'** is hours and has a range of 00 through 23 and **‘mm'** is minutes and has a range of 00

through 59. The minimum interval that can be specified is 10 minutes. **‘hh'** must be specified even if zero. For example, the following specification indicates that the data set is to be spun at 20 minute intervals:

```
SPIN=(UNALLOC,'+00:20')
```

The time interval must be specified within apostrophes.

nnn

nnnK

nnnM Data set will be spun when it has **nnn** lines, where **nnn** is the number of lines and must start with a minimum of 500, **K** is thousands, and **M** is millions.

NOCMND
Data set cannot be spun before it is unallocated.

CMNDONLY
Data set will only be spun when an operator issues a command to spin the data set.

Example: To specify a SPIN interval of 999 lines, code:

```
KEY  #    LEN  PARM
802A 0001 0003 F9F9F9
```

SYMLIST on a DD statement - Key = '802B'

DALSYML lists the symbol names to be passed to JES when an internal reader is allocated. This key is allowed only when text unit DALSYSOU is also coded, and its value must be the 6-character string `INTRDR`. This is because it is meant for allocations to an internal reader only.

When you code this key, # contains the number of symbol names being specified up to a maximum of 128 (X'0080'), LEN is the length of the PARM, and PARM contains valid symbol names. In addition, '*' is a valid value for a symbol name to indicate that all symbols can be used in JCL; refer to the SYMLIST keyword in the DD statement in *z/OS MVS JCL Reference*.

Example: To specify SYMLIST with symbol name NAMES on a DD, code:

```
KEY    #      LEN    PARM
802B   0001   0005   D5 C1 D4 C5 E2
0018   0001   0001   5C
0019   0001   0006   C9 D5 E3 D9 C4 D9
```

The DSNTYPE text unit key section has a format for multiple text units.

DSNTYPE version - Key = '802C'

DALDSNV specifies the data set type version information.

When you code DALDSNV, # must be 1, LEN must be 1, and the PARM field contains the DSNTYPE version number. For more information, refer to the DSNTYPE=LIBRARY keyword in the DD statement in *z/OS MVS JCL Reference*.

Example: To specify the data set type (DALDSNT) and version (DALDSNV) for the DD statement with name DD1, code:

```
KEY    #      LEN    PARM
0001   0001   0003   C4 C4 C1
8012   0001   0001   80
802C   0001   0001   02
```

In this example, the data set type represents a library (PDSE) version 2.

Note: DALDSNV is meaningless without the proper DALDSNT value. Be sure to specify both when using DALDSNV.

Maximum PDSE Generation - Key = '802D'

When you have applied APAR OA42358, DALMAXG specifies the maximum number of PDSE member generations to be retained by the system.

When you code DALMAXG, # must be 1, LEN must be 4 and the PARM field contains a value between 1 and 2,000,000,000.

Example: To specify 255 as the maximum number of PDSE member generations (DALMAXG) for the DD statement with name DD1, code:

```
Key   #     LEN   PARM
0001  0001  0003  C4 C4 C1
802D  0001  0004  000000FF
```

In this example, the data set type represents a library (PDSE) version 2.

Note: DALMAXG is relevant only for data sets of type LIBRARY. You may need to specify DALDSNT to set the DSNTYPE.

For more information, refer to *z/OS Summary of Message and Interface Changes*.

GDGORDER specification – Key = '802E'

For a DD that specifies the base name of a GDG data set, indicating a GDG ALL request, this text unit specifies the order in which the individual generation data sets (GDSs) will be concatenated.

When you code GDGORDER, # must be one, LEN must be the length of the PARM value (1), and PARM contains one of the following values:

x'80' USECATLG – the GDS concatenation will be ordered as specified in the GDG data set catalog entry.

X'40' LIFO – the GDS concatenation is ordered with the newest GDS defined first and the oldest GDS last. This is the default behavior when FIFO or LIFO is not specified on the IDCAMS utility when defining the GDG data set, and in releases prior to z/OS V2R1.

X'20' FIFO – the GDS concatenation is ordered with the oldest GDS defined first and the newest GDS last.

Example: To specify that the GDSs of a GDG data set request be allocated with oldest data first, code:

```
Key   #     LEN   PARM
802E  0001  0001  20
```

JCL DD statement DCB subparameters and equivalent text units

Use the following table to convert JCL DD statement DCB parameters into text units. The table lists JCL DD statement DCB subparameters in alphabetical order, along with the equivalent verb code 01 text unit key and mnemonic. Some DCB subparameters have multiple text unit keys associated with them. In these cases, the same DCB subparameter is repeated in the table for each text unit key associated with it.

The following JCL statement DD DCB subparameters do not have equivalent text units and do not appear in the table:

- CYLOFL
- NTM
- RKP.

The following text unit keys are obsolete. The system checks these text units for syntax and ignores them. They appear in Table 86 on page 666 and are described in the text unit descriptions for your reference when using old code, but they are not included in this table.

- DALBUFRQ (Key X'0037')
- DALCODE (Key X'0039')
- DALSOWA (Key X'004C')
- DALFRID (Key X'0058').

Table 85. JCL DD Statement DCB Subparameters and Equivalent Text Units

JCL DD Statement DCB Subparameter	Text Unit Mnemonic	Text Unit Key	Described in:
BFALN	DALBFALN	002E	“BFALN specification - Key = '002E'” on page 667
BFTEK	DALBFTEK	002F	“BFTEK specification - Key = '002F'” on page 667
BLKSIZE	DALBLKSZ	0030	“BLKSIZE specification - Key = '0030'” on page 668
BUFIN	DALBUFIN	0031	“BUFIN specification - Key = '0031'” on page 668
BUFL	DALBUFL	0032	“BUFL specification - Key = '0032'” on page 668
BUFMAX	DALBUFMX	0033	“BUFMAX specification - Key = '0033'” on page 668
BUFNO	DALBUFNO	0034	“BUFNO specification - Key = '0034'” on page 668
BUFFOFF	DALBUFOF	0035	“BUFFOFF specification - Key = '0035'” on page 669
BUFFOUT	DALBUFOU	0036	“BUFOUT specification - Key = '0036'” on page 669
BUFSIZE	DALBUFSZ	0038	“BUFSZ specification - Key = '0038'” on page 669
CPRI	DALCPRI	003A	“CPRI specification - Key = '003A'” on page 670
DEN	DALDEN	003B	“DEN specification - Key = '003B'” on page 670
DIAGNS	DALDIAGN	0054	“Diagnostic trace specification (DIAGNS=TRACE) - Key = '0054'” on page 676
DSORG	DALDSORG	003C	“DSORG specifications - Key = '003C'” on page 670
EROPT	DALEROPT	003D	“EROPT specification - Key = '003D'” on page 671
FUNC	DALFUNC	005A	“FUNC= specification - Key = '005A'” on page 676
GNCP	DALGNCP	003E	“GNCP specification - Key = '003E'” on page 671
INTVL	DALINTVL	003F	“INTVL specification - Key = '003F'” on page 671
KEYLEN	DALKYLEN	0040	“KEYLEN specification - Key = '0040'” on page 672
LIMCT	DALLIMCT	0041	“LIMCT specification - Key = '0041'” on page 672
LRECL=bytes	DALLRECL	0042	“LRECL specification - Key = '0042'” on page 672
LRECL=nnnnnK	DALLRECK	006B	“Record length specification - Key = '006B'” on page 677
MODE	DALMODE	0043	“MODE specification - Key = '0043'” on page 672
NCP	DALNCP	0044	“NCP specification - Key = '0044'” on page 672
OPTCD	DALOPTCD	0045	“OPTCD specification - Key = '0045'” on page 673

Table 85. JCL DD Statement DCB Subparameters and Equivalent Text Units (continued)

JCL DD Statement DCB Subparameter	Text Unit Mnemonic	Text Unit Key	Described in:
PCI=receiving PCI	DALPCIR	0046	“Receiving PCI specification - Key = '0046'” on page 673
PCI=sending PCI	DALPCIS	0047	“Sending PCI specification - Key = '0047'” on page 674
PRTSP	DALPRTSP	0048	“PRTSP specification - Key = '0048'” on page 674
RECFM	DALRECFM	0049	“RECFM specification - Key = '0049'” on page 674
RESERVE=bytes1	DALRSRVF	004A	“First buffer reserve specification - Key = '004A'” on page 675
RESERVE=bytes2	DALRSRVS	004B	“Secondary buffer reserve specification - Key = '004B'” on page 675
STACK	DALSTACK	004D	“STACK specification - Key = '004D'” on page 675
THRESH	DALTHRSH	004E	“THRESH specification - Key = '004E'” on page 675
TRTCH	DALTRTCH	004F	“TRTCH specification – Key = '004F'” on page 676

DCB attribute text units

Use verb code 01 and the text unit keys listed in Table 86 and described on the following pages to specify the DCB attributes of the data set being dynamically allocated. These attributes are described in *z/OS MVS JCL Reference* under the DCB parameter, and in *z/OS DFSMS Macro Instructions for Data Sets*.

Table 86. –Verb Code 01 (DCB Attributes) Text Unit Keys, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	DYNALLOC Function
002E	DALBFALN	Specifies buffer alignment.
002F	DALBFTEK	Specifies the buffering technique.
0030	DALBLKSZ	Specifies blocksize.
0031	DALBUFIN	Specifies the receiving buffer count.
0032	DALBUFL	Specifies the buffer length.
0033	DALBUFMX	Specifies the buffer count per line.
0034	DALBUFNO	Specifies the buffer count per DCB.
0035	DALBUFOF	Specifies the buffer offset.
0036	DALBUFOU	Specifies the sending buffer count.
0037	DALBUFRQ	Obsolete. The system checks the text unit for syntax and ignores it.
0038	DALBUFSZ	Specifies the line group buffer size.
0039	DALCODE	Obsolete. The system checks the text unit for syntax and ignores it.
003A	DALCPRI	Specifies the relative sending and receiving priority.
003B	DALDEN	Specifies the magnetic tape density.
003C	DALDSORG	Specifies the data set organization.
003D	DALEROPT	Specifies reading and writing error options.
003E	DALGNCP	Specifies the GAM-I/O count per WAIT macro.
003F	DALINTVL	Specifies the line polling interval per group.
0040	DALKYLEN	Specifies the data set key lengths.
0041	DALLIMCT	Specifies the search limit.
0042	DALLRECL	Specifies the logical record length.
0043	DALMODE	Specifies card punch/reader operational mode.
0044	DALNCP	Specifies the READ/WRITE count per CHECK.
0045	DALOPTCD	Specifies the control program's operational services.
0046	DALPCIR	Specifies the relationship of the receiving PCI to the allocation and freeing of buffers.

Table 86. –Verb Code 01 (DCB Attributes) Text Unit Keys, Mnemonics, and Functions (continued)

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0047	DALPCIS	Specifies the relationship of the sending PCI to the allocation and freeing of buffers.
0048	DALPRTSP	Specifies printer line spacing.
0049	DALRECFM	Specifies the record format.
004A	DALRSRVF	Specifies the first buffer's reserve byte count for insertion of data.
004B	DALRSRVS	Specifies the secondary buffer's reserve byte count for insertion of data.
004C	DALSOWA	Obsolete. The system checks the text unit for syntax and ignores it.
004D	DALSTACK	Specifies the card punch's stacker bin.
004E	DALTHRSH	Specifies the use percentage of non-reusable direct access message queue records per flush closedown.
004F	DALTRTCH	Specifies the 7-track tape recording technique.
0054	DALDIAGN	Requests OPEN/CLOSE/EOV diagnostic trace option.
005A	DALFUNC	Specifies the type of data set to be opened for the 3525 Card-Read-Punch-Print.
005B	DALFRID	Obsolete. The system checks the text unit for syntax and ignores it.
006B	DALLRECK	Specifies that a record length is in increments of 1,024 (K).
0072	DALKILO	Specifies a block size type of Kilobyte.
0073	DALMEG	Specifies a block size type of Megabyte.
0074	DALGIG	Specifies a block size type of Gigabyte.

BFALN specification - Key = '002E'

DALBFALN specifies the buffer alignment. It is mutually exclusive with the GAM-I/O count key (DALGNCP). When you code DALBFALN, # and LEN must be one, and PARM contains one of the following values:

X'01' Fullword not a doubleword boundary (F)

X'02' Doubleword boundary (D)

Example: To specify doubleword boundary, code:

```
KEY    #      LEN    PARM
002E   0001   0001   02
```

BFTEK specification - Key = '002F'

DALBFTEK specifies the buffering technique to be used. It is mutually exclusive with the GAM-I/O count key (DALGNCP). When you code DALBFTEK, # and LEN must be one, and PARM contains one of the following values:

X'08' Dynamic buffering (D)

X'10' Exchange buffering (E)

X'20' Record buffering (R)

X'40' Simple buffering (S)

X'60' Record area buffering (A)

Example: To specify exchange buffering, code:

```
KEY    #      LEN    PARM
002F   0001   0001   10
```

BLKSIZE specification - Key = '0030'

DALBLKSZ specifies the block size. It is mutually exclusive with the buffer size key (DALBUFSZ). When you code DALBLKSZ, # must be one, LEN must be two or eight, and PARM contains the block size. The maximum PARM value is '7FF8' (32,760) for DASD and '80000000' (2,147,483,648) for tape. A LEN value of eight must be used for block size specifications greater than '7FF8'. You may also code one of the block size type keys (DALKILO, DALMEG or DALGIG) when you specify DALBLKSZ. If you do not specify one of those keys, the value coded on DALBLKSZ is considered to be in bytes.

Example: To specify a block size of 80, code:

```
KEY    #      LEN    PARM
0030   0001   0002   00 50
```

or

```
KEY    #      LEN    PARM
0030   0001   0008   00 00 00 00 00 00 00 50
```

BUFIN specification - Key = '0031'

DALBUFIN specifies the number of buffers to be initially assigned for receiving operations for each line in the line group. It is mutually exclusive with the buffer number (DALBUFNO) and buffer request (DALBUFRQ) keys. When you code DALBUFIN, # and LEN must be one, and PARM contains the number of buffers. The maximum PARM value is '0F' (15).

Example: To specify 2 buffers, code:

```
KEY   #     LEN   PARM
0031  0001  0001  02
```

BUFL specification - Key = '0032'

DALBUFL specifies the buffer length. When you code this key, # must be one, LEN must be two, and PARM contains the buffer length. The maximum PARM value is '7FF8' (32,760).

Example: To specify a buffer length of 80, code:

```
KEY    #      LEN    PARM
0032   0001   0002   00 50
```

BUFMAX specification - Key = '0033'

DALBUFMX specifies the maximum number of buffers to be allocated to a line at one time. It is mutually exclusive with the NCP key (DALNCP). When you code DALBUFMX, # and LEN must be one, and PARM contains the number of buffers. The maximum PARM value is '0F' (15).

Example: To specify 4 buffers, code:

```
KEY    #      LEN    PARM
0033   0001   0001   04
```

BUFNO specification - Key = '0034'

DALBUFNO specifies the number of buffers to be assigned to the data control block. It is mutually exclusive with the BUFIN (DALBUFIN), BUFOUT (DALBUFOU), and BUFRQ (DALBUFRQ) keys. When you code DALBUFNO, # and LEN must be one, and PARM contains the number of buffers.

Example: To specify 2 buffers, code:

```
KEY   #     LEN   PARM
0034  0001  0001  02
```

BUFFOFF specification - Key = '0035'

DALBUFOF specifies the buffer offset. When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'80' The block prefix is four bytes long and contains the block length (L)

X'nn' The length of the block prefix (maximum of X'63' (99))

Example: To specify an offset of 16, code:

```
KEY    #      LEN    PARM
0035   0001   0001   10
```

BUFOUT specification - Key = '0036'

DALBUFOU specifies the number of buffers to be assigned initially for sending operations for each line in the group. It is mutually exclusive with the BUFNO (DALBUFNO) and BUFRQ (DALBUFRQ) keys. When you code DALBUFOU, # and LEN must be one, and PARM contains the number of buffers. The maximum PARM value is '0F' (15).

Example: To specify 4 buffers, code:

```
KEY    #      LEN    PARM
0036   0001   0001   04
```

BUFRQ specification - Key = '0037'

DALBUFRQ specifies the number of buffers to be requested in advance for the GET macro. It is mutually exclusive with the BUFNO (DALBUFNO), BUFIN (DALBUFIN), and BUFOUT (DALBUFOU) keys. When you code DALBUFRQ, # and LEN must be one, and PARM contains the number of buffers.

Example: To specify 4 buffers, code:

```
KEY    #      LEN    PARM
0037   0001   0001   04
```

BUFSZ specification - Key = '0038'

DALBUFSZ specifies the length in bytes of each of the buffers to be used for all lines in a particular line group. It is mutually exclusive with the block size key (DALBLKSZ). When you code DALBUFSZ, # must be one, LEN must be two, and PARM contains the buffer length.

Example: To specify a buffer length of 80, code:

```
KEY    #      LEN    PARM
0038   0001   0002   00 50
```

CODE specification - Key = '0039'

DALCODE specifies the paper tape code in which the data is punched. It is mutually exclusive with the key length (DALKYLEN), MODE (DALMODE), printer spacing (DALPRTSP), STACK (DALSTACK), and TRTCH (DALTRTCH) keys. When you code DALCODE, # and LEN must be one, and PARM contains one of the following values:

X'02' Teletype 5-track (T)

X'04' USASCII 8-track (A)

X'08' National Cash Register 8-track (C)

X'10' Burroughs 7-track (B)

X'20' Friden 8-track (F)

X'40' IBM BCD 8-track (I)

X'80' No conversion (N)

Example: To specify USASCII, code:

```
KEY    #      LEN    PARM
0039   0001   0001   04
```

CPRI specification - Key = '003A'

DALCPRI specifies the relative priority to be given to sending and receiving operations. It is mutually exclusive with the THRESH key (DALTHRSH). When you code DALCPRI, # and LEN must be one, and PARM contains one of the following values:

X'01' Send priority (S)

X'02' Equal priority (E)

X'04' Receiving priority (R)

Example: To specify equal priority, code:

```
KEY    #      LEN    PARM
003A   0001   0001   02
```

DEN specification - Key = '003B'

DALDEN specifies the magnetic tape density. When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'03' 200 bpi 7-track (0)

X'43' 556 bpi 7-track (1)

X'83' 800 bpi 7-track, 800 bpi 9 - track (2)

X'C3' 1600 bpi 9-track (3)

X'D3' 6250 bpi 9-track (4)

Example: To specify 1600 bpi 9 - track, code:

```
KEY    ##     LEN    PARM
003B   0001   0001   C3
```

DSORG specifications - Key = '003C'

DALDSORG specifies the data set organization. When you code this key, # must be one, LEN must be two, and PARM contains one of the following values:

X'0008'
VSAM

X'0080'
Graphics (GS)

X'0200'
Partitioned organization (PO)

X'0300'
Partitioned organization unmovable (POU)

X'0400'
Government of message transfer to or from a telecommunications message processing queue (MQ)

X'0800'
Direct access message queue (CQ)

X'1000'
Communication line group (CX)

X'2000'
Direct access (DA)

X'2100'
Direct access unmovable (DAU)

X'4000'
Physical sequential (PS)

X'4100'
Physical sequential unmovable (PSU)

Example: To specify Partitioned Organization, code:

```
KEY    #      LEN    PARM
003C   0001   0002   02 00
```

EROPT specification - Key = '003D'

DALEROPT specifies the option to be executed if an error occurs in writing or reading a record. When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'10' Online BSAM testing (T)

X'20' To cause abnormal end of task (ABE)

X'40' To skip the block causing the error (SKP)

X'80' To accept the block causing the error (ACC)

Example: To specify the SKP error option, code:

```
KEY    #      LEN    PARM
003D   0001   0001   40
```

GNCP specification - Key = '003E'

DALGNCP specifies the maximum number of GAM input/output macros that will be issued before a WAIT macro is issued. It is mutually exclusive with the BFTEK (DALBFTK) and BFALN (DALBFAL) keys. When you code DALGNCP, # and LEN must be one, and PARM contains the GNCP value. The maximum PARM value is '63' (99).

Example: To specify a GNCP value of four, code:

```
KEY    #      LEN    PARM
003E   0001   0001   04
```

INTVL specification - Key = '003F'

DALINTVL specifies the polling interval for the lines in the line group. This key is mutually exclusive with the UCS (DALUCS) and FCB (DALFCB) keys. When you code this key, # and LEN must be one, and PARM contains the INTVL value.

Example: To specify an INTVL value of 10, code:

```
KEY    #      LEN    PARM
003F   0001   0001   0A
```

KEYLEN specification - Key = '0040'

DALKYLEN specifies the length, in bytes, of the keys used in the data set. It is mutually exclusive with the CODE (DALCODE), MODE (DALMODE), PRTSP (DALPRTSP), STACK (DALSTACK), and TRTCH (DALTRTCH) keys. When you code this key, # and LEN must be one, and PARM contains the key length.

Example: To specify a key length of eight, code:

```
KEY    #      LEN    PARM
0040   0001   0001   08
```

LIMCT specification - Key = '0041'

DALLIMCT specifies the search limit. When you code this key, # must be one, LEN must be three, and PARM contains the search limit value. The maximum PARM value is '007FF8' (32,760).

Example: To specify a search limit of 1000, code:

```
KEY    #      LEN    PARM
0041   0001   0003   0003E8
```

LRECL specification - Key = '0042'

DALLRECL specifies the actual or maximum length, in bytes, of a logical record. When you code this key, # must be one, LEN must be two, and PARM contains one of the following values:

X'8000'
Variable length spanned records processed under QSAM and BSAM, the logical records exceed 32,756 bytes (X)

X'nnnn'
The logical record length. The maximum value for nnnn is '7FF8' (32,760).

Example: To specify a logical record length of 80, code:

```
KEY    #      LEN    PARM
0042   0001   0002   0050
```

MODE specification - Key = '0043'

DALMODE specifies the mode of operation for a card reader or punch. It is mutually exclusive with the CODE (DALCODE), KEYLEN (DALKYLEN), PRTSP (DALPRTSP), and TRTCH (DALTRTCH) keys. When you code DALMODE, # and LEN must be one, and PARM contains one of the following values:

X'40'	EBCDIC mode (E)
X'50'	EBCDIC, read column eliminate mode (ER)
X'60'	EBCDIC, optical mark read mode (EO)
X'80'	Card image mode (C)
X'90'	Card image, read column eliminate mode (CR)
X'A0'	Card image, optical mark read mode (CO)

Example: To specify EBCDIC mode, code:

```
KEY    #      LEN    PARM
0043   0001   0001   40
```

NCP specification - Key = '0044'

DALNCP specifies the maximum number of READ or WRITE macros issued before a CHECK macro is issued. It is mutually exclusive with the BUFMAX

(DALBUFMX) key. When you code DALNCP, # and LEN must be one, and PARM contains the NCP value. The maximum PARM value is X'FF' (255).

Example: To specify an NCP value of two, code:

```
KEY    #      LEN    PARM
0044   0001   0001   02
```

OPTCD specification - Key = '0045'

DALOPTCD specifies optional services to be performed. When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'01' Relative block addressing (R), or to select character arrangement tables for the 3800 printer (J)

X'02' User totaling facility (T)

X'04' Reduced tape error recovery or direct DASD search (Z)

X'08' Direct addressing (A), or translation of ASCII to or from EBCDIC (Q)

X'10' Feedback (F), or hopper-empty exit (H), or online correction for optical readers (O)

X'40' Disregarding end-of-file recognition for tape (B), or allowance of data checks caused by an invalid character.

X'80' Write validity check.

Note: When you are specifying more than one OPTCD value, PARM contains the **sum** of the values.

For more information regarding the OPTCD specification key, see *z/OS DFSMS Macro Instructions for Data Sets*.

Example: To specify OPTCD value B, code:

```
KEY    #      LEN    PARM
0045   0001   0001   40
```

Example: To specify OPTCD values B and Z, code:

```
KEY    #      LEN    PARM
0045   0001   0001   44
```

Receiving PCI specification - Key = '0046'

DALPCIR specifies the relationship of program-controlled interrupts (PCI) during receiving operations to the allocation and freeing of buffers. When you code DALPCIR, # and LEN must be one, and PARM contains one of the following values:

X'02' A PCI and no new buffer allocated (R)

X'08' No PCIs (N)

X'20' A PCI and new buffer allocated (A)

X'80' A PCI, new buffer allocated, and the first buffer remains allocated (X)

Example: To specify no PCIs during receiving operations, code:

```
KEY    #      LEN    PARM
0046   0001   0001   08
```

Sending PCI specification - Key = '0047'

DALPCIS specifies the relationship of PCIs during sending operations to the allocation and freeing of buffers. When you code DALPCIS, # and LEN must be one, and PARM contains one of the following values:

X'01' A PCI and no new buffer allocated (R)

X'04' No PCIs (N)

X'10' A PCI and a new buffer allocated (A)

X'40' A PCI, new buffer allocated, and first buffer remains allocated (X)

Example: To specify no PCIs during sending operations, code:

```
KEY    #      LEN    PARM
0047   0001   0001   04
```

PRTSP specification - Key = '0048'

DALPRTSP specifies printer line spacing. It is mutually exclusive with the CODE (DALCODE), KEYLEN (DALKYLEN), MODE (DALMODE), STACK (DALSTACK), and TRTCH (DALTRTCH) keys. When you code DALPRTSP, # and LEN must be one, and PARM contains one of the following values:

X'01' No spacing (0)

X'09' One-line spacing (1)

X'11' Two-line spacing (2)

X'19' Three-line spacing (3)

Example: To specify no spacing, code:

```
KEY    #      LEN    PARM
0048   0001   0001   01
```

RECFM specification - Key = '0049'

DALRECFM specifies the record format. When you code this key, # and LEN must be one, and PARM contains one of the following values:

X'02' Machine code printer control characters in record (M), or complete QTAM record (R)

X'04' ASA printer control characters in record (A), or complete QTAM message (G)

X'08' Standard fixed records, spanned variable records, or segment of QTAM message (S)

X'10' Blocked records (B)

X'20' Variable ASCII records (D), or track overflow (T)

X'40' Variable records (V)

X'80' Fixed records (F)

X'C0' Undefined records (U)

Note: When you code combinations of RECFM values, PARM contains the **sum** of the values.

Example: To specify fixed records, code:

```
KEY    #      LEN    PARM
0049   0001   0001   80
```

Example: To specify variable blocked (VB) records, code:

```
KEY    #      LEN    PARM
0049   0001   0001   50
```

First buffer reserve specification - Key = '004A'

DALRSRVF specifies the number of bytes to be reserved in the first buffer for insertion of data by the DATETIME and SEQUENCE macros. The UCS (DALUCS) key is mutually exclusive with DALRSRVF. When you code this key, # and LEN must be one, and PARM contains the number of bytes to reserve.

Example: To reserve 8 bytes in the first buffer, code:

```
KEY    #      LEN    PARM
004A   0001   0001   08
```

Secondary buffer reserve specification - Key = '004B'

DALRSRVS specifies the number of bytes to be reserved in buffers other than the first for insertion of data by the DATETIME and SEQUENCE macros. The UCS (DALUCS) key is mutually exclusive with DALRSRVS. When you code this key, # and LEN must be one, and PARM contains the number of bytes to reserve.

Example: To reserve 8 bytes in secondary buffers, code:

```
KEY    #      LEN    PARM
004B   0001   0001   08
```

SOWA specification - Key = '004C'

DALSOWA specifies the size, in bytes, of the user-provided input work areas for telecommunication jobs. When you code this key, # must be one, LEN must be two, and PARM contains the number of bytes. The maximum PARM value is '7FF8' (32,760).

Example To specify a 256-byte work area, code:

```
KEY    #      LEN    PARM
004C   0001   0002   0100
```

STACK specification - Key = '004D'

DALSTACK specifies the stacker bin to receive cards. The CODE (DALCODE), KEYLEN (DALKYLEN), PRTSP (DALPRTSP), and TRTCH (DALTRTCH) keys are mutually exclusive with DALSTACK. When you code this key, # and LEN are one, and PARM contains one of the following values:

X'01'	Bin 1 (1)
X'02'	Bin 2 (2)

Example: To specify stacker 2, code:

```
KEY    #      LEN    PARM
004D   0001   0001   02
```

THRESH specification - Key = '004E'

DALTHRSH specifies the percentage of non-reusable disk message queue records to be used before a flush closedown occurs. The CPRI (DALCPRI) key is mutually exclusive with DALTHRSH. When you code this key, # and LEN must be one, and PARM contains the percentage. The maximum PARM value is '64' (100).

Example: To specify a THRESH percentage of 99, code:

```
KEY    #      LEN    PARM
004E   0001   0001   63
```

TRTCH specification – Key = '004F'

DALTRTCH specifies the recording technique for 7-track tape. It is mutually exclusive with the CODE (DALCODE), KEYLEN (DALKYLEN), MODE (DALMODE), PRTSP (DALPRTSP), and STACK (DALSTACK) keys. When you code DALTRTCH, # and LEN must be one, and PARM contains one of the following values:

X'04'	Non-compaction mode
X'08'	Compaction mode
X'13'	Data conversion (C)
X'23'	Even parity (E)
X'2B'	Even parity and BCD/EBCDIC translation (ET)
X'3B'	BCD/EBCDIC translation (T)

Example: To specify even parity, code:

```
KEY    #      LEN    PARM
004F   0001   0001   23
```

Diagnostic trace specification (DIAGNS=TRACE) - Key = '0054'

DALDIAGN requests the OPEN/CLOSE/EOV trace option, which gives a module-by-module trace of OPEN/CLOSE/EOV's work area and the user's DCB. When you code DALDIAGN, # must be zero; LEN and PARM are not specified. GTF must be active in the system while the job that requested the trace is running.

Example: To specify the diagnostic trace specification, code:

```
KEY    #      LEN   PARM
0054   0000   -     -
```

FUNC= specification - Key = '005A'

DALFUNC can be used with BSAM and QSAM; it specifies the type of data set to be opened for the 3525 Card Read-Punch-Print. When you code DALFUNC, # and LEN must be one, and PARM contains one of the following values:

X'10'	W
X'12'	WT
X'14'	WX
X'16'	WXT
X'20'	P
X'30'	PW
X'34'	PWX
X'36'	PWXT
X'40'	R
X'50'	RW
X'52'	RWT
X'54'	RWX
X'56'	RWXT
X'60'	RP

X'68'	RPD
X'70'	RPW
X'74'	RPWX
X'76'	RPWXT
X'78'	RPWD
X'80'	I

Where:

D	is data protection for a punch data set
I	is interpret punch data set
P	is punch
R	is read
T	is two line printer
W	is print
X	is printer

Note:

1. In the absence of this information, the system assumes P.
2. D, X, and T cannot be coded alone.
3. If you specify D as part of a value, you must also specify the FCB image-id key (DALFCBIM), giving the image identifier for the data protection image.

Example: To specify FUNC=RPWD, code:

```
KEY    #      LEN    PARM
005A   0001   0001   78
```

FRID= specification - Key = '005B'

DALFRID specifies the last four characters of a SYS1.IMAGELIB member name to be used in the interpretation of documents for input to the IBM 3886 character reader. The FCB (DALFCBIM) key is mutually exclusive with DALFRID.

When you code DALFRID, # must be one, LEN is the number of characters specified, and PARM contains the characters of the IMAGELIB member name. The characters must be alphanumeric or national. If the length of the member name is four or less, code the entire name.

Example: To specify the last four characters of member name SHARK1, code:

```
KEY    #      LEN    PARM
005B   0001   0004   C1 D9 D2 F1
```

Record length specification - Key = '006B'

DALLRECK indicates that the record length of an ISO/ANSI/FIPS Version 3 tape data set is specified in increments of 1,024 (K).

When you code DALLRECK, # must be zero; LEN and PARM are not specified. You code DALLRECK only if specifying DALLRECL. When DALLRECL is specified with DALLRECK, the maximum value of DALLRECL is 16,383 (3FFF). See *z/OS MVS JCL Reference* for the restrictions regarding the use of LRECL for nnnnnK.

Example: To specify a record length of 80K, code:

```
KEY    #      LEN   PARM
006B   0000   -     -
```

Kilobyte BLKSIZE type specification - Key = '0072'

DALKILO specifies a block size in kilobytes. (1 kilobyte = 1024 bytes.) It is mutually exclusive with the megabyte and gigabyte block size keys (DALMEG and DALGIG), and is used to specify that the block size requested on DALBLKSZ is in kilobytes. When you specify DALKILO, you must also code the DALBLKSZ key. When you specify DALKILO, the maximum PARM value for DALBLKSZ is '00200000'. When you code DALKILO, # must be zero, and you do not specify LEN and PARM.

Example: To specify block size in kilobytes, code:

```
KEY    #      LEN    PARM
0072   0000    -       -
```

Megabyte BLKSIZE type specification - Key = '0073'

DALMEG specifies a block size in megabytes. (1 megabyte = 1024 kilobytes.) It is mutually exclusive with the kilobyte and megabyte block size type keys (DALKILO, DALGIG), and is used to specify that the block size requested via DALBLKSZ is in megabytes. When you specify DALMEG, the maximum PARM value for DALBLKSZ is '0080'. When you code DALMEG, # must be zero, and you do not specify LEN and PARM.

Example: To specify block size in megabytes, code:

```
KEY    #      LEN    PARM
0073   0000    -       -
```

Gigabyte BLKSIZE type specification - Key = '0074'

DALGIG specifies a block size in gigabytes. (1 gigabyte = 1024 megabytes.) It is mutually exclusive with the kilobyte and megabyte block size type keys (DALKILO and DALMEG), and is used to specify that the block size requested on DALBLKSZ is in gigabytes. When you specify DALGIG, you must also code the DALBLKSZ key. When you specify DALGIG, the maximum PARM value for DALBLKSZ is '0002'. When you code DALGIG, # must be zero, and you do not specify LEN and PARM.

Example: To specify block size in gigabytes, code:

```
KEY    #      LEN    PARM
0074   0000    -       -
```

Non-JCL dynamic allocation functions

The keys listed in Table 87 on page 679 and described on the following pages do not have JCL equivalents; they have meaning only when you request dynamic allocation by dsname (verb code 01). You can request that the system return information about either a dsname or a ddname dynamic allocation request by specifying either of the following:

- The non-JCL dynamic allocation functions keys 0055-0057 and 005D, which request that the system return information about your dsname allocation request
- Information retrieval keys greater than C000, described in “Dynamic information retrieval text units” on page 697

For either verb code 01 or verb code 07 information retrieval, the data returned in the PARM field is valid only for the number of bytes returned in the LEN field. If on input the PARM field is longer than the information returned on output, the remaining bytes in PARM might contain data from previous invocations of DYNALLOC. The information retrieval function does not pad the remaining bytes with blanks.

For example, on **Input** the text unit initializes the PARM field to "AAAAAAAA". On **Output** the text unit contains the name "MYDD" and a LEN of X'0004'. Beyond the valid length (4 bytes) of "MYDD", the PARM field might contain data from the previous request ("MYDDAAAA").

```
        KEY  #     LEN    PARM
Input:  0055 0001  0008   C1C1C1C1C1C1C1C1

Output: 0055 0001  0004   D4E8C4C4C1C1C1C1
```

When you request information for multiple parameters, the LEN and PARM fields of the next parameter immediately follow the information returned from the previous parameter. For example, on **Output** the text unit below contains a second LEN and PARM pair. The request returns the information "X" in the PARM2 field. The information in the fields is **contiguous**; spaces are shown in the example only for readability.

```
        KEY  #     LEN1   PARM1     LEN2     PARM2
Output: nnnn 0002  0004   D4E8C4C4  0001     E7
```

Table 87. Verb Code 01 (Non-JCL Dynamic Allocation Functions) – Text Units, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0050	DALPASSW	Specifies the password for a protected data set.
0052	DALPERMA	Specifies the permanently allocated attribute.
0053	DALCNVRT	Specifies the convertible attribute.
0055	DALRTDDN	Requests the return of the associated ddname.
0056	DALRTDSN	Requests the return of the allocated data set's name.
0057	DALRTORG	Requests the return of data set organization.
005C	DALSSREQ	Specifies allocation of a subsystem data set.
005D	DALRTVOL	Requests the return of the volume serial number.
0062	DALSSATT	Specifies allocation of a subsystem data set to SYSIN.
006E	DALBRTKN	Specifies a spool data set browse token that contains information about a JES2 spool data set that a user asks to browse.
006F	DALINCHG	Specifies the tape recording technique and media type to use for a new data set request.
0075	DALUASSR	Specifies allocation of a subsystem data set.
0077	DALUNQDS	Specifies a uniquely allocated temporary data set.

Password specification - Key = '0050'

DALPASSW specifies the password for a password-protected data set. The dsname key (DALDSNAM) is required with this key. When you code DALPASSW, # must be one, LEN contains the length of the password, up to a maximum of 8, and PARM contains the password.

Example: To specify the password, MYKEY, code:

```
KEY    #      LEN    PARM
0050   0001   0005   D4 E8 D2 C5 E8
```

Permanently allocated attribute specification - Key = '0052'

DALPERMA specifies that the permanently allocated attribute is to be assigned to this allocation. (See “Permanently allocated attribute” on page 540 for a description of the permanently allocated attribute.) When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify assignment of the permanently allocated attribute, code:

```
KEY    #      LEN   PARM
0052   0000   -     -
```

Convertible attribute specification - Key = '0053'

DALCNVRT specifies that the convertible attribute is to be assigned to this allocation. (See “Convertible attribute” on page 540 for a description of the convertible attribute.)

Note: This specification is the default if the permanently allocated attribute key (DALPERMA) is not coded.

When you code DALCNVRT, # must be zero; LEN and PARM are not specified.

Example: To specify assignment of the convertible attribute, code:

```
KEY    #      LEN   PARM
0053   0000   -     -
```

Ddname return specification - Key = '0055'

DALRTDDN requests that the ddname associated with the allocation be returned to the caller of DYNALLOC. When you code DALRTDDN, # must be one, and LEN must be at least the length of the ddname, and can be longer, up to a maximum of 255 characters. The PARM field must be the length specified by the value in LEN. Dynamic allocation places the allocated ddname in PARM and updates LEN to the length of the returned ddname.

The system does not fail the dynamic allocation request if it cannot obtain the ddname. Therefore, when you use this key, provide an SVC 99 request block extension (see “Setting up the request block extension” on page 555). Upon return, verify that fields S99EERR and S99EINFO in the request block extension are zero. If these fields are non-zero, do not use the returned ddname.

Example: To request that the allocated ddname be returned, code:

```
KEY    #      LEN    PARM
0055   0001   0008   --------
```

This specification would be updated upon the assignment of the ddname DD1 as follows:

```
KEY    #      LEN    PARM
0055   0001   0003   C4 C4 F1-----
```

Dsname return specification - Key = '0056'

DALRTDSN requests that the dsname that is allocated be returned to the caller of DYNALLOC. When you code DALRTDSN, # must be one, and LEN must be at least the length of the dsname, and can be longer, up to a maximum of 255 characters. The PARM field must be the length specified by the value in LEN. Dynamic allocation places the allocated dsname in PARM and updates LEN to the length of the returned dsname.

The system does not fail the dynamic allocation request if it cannot obtain the dsname. Therefore, when you use this key, provide an SVC 99 request block extension (see “Setting up the request block extension” on page 555). Upon return, verify that fields S99EERR and S99EINFO in the request block extension are zero. If these fields are non-zero, do not use the returned dsname.

Example: To request that the allocated dsname be returned, code:

```
KEY    #      LEN    PARM
0056   0001   002C   --------...--
```

This specification would be updated for the allocation of the dsname ABC as follows:

```
KEY    #      LEN    PARM
0056   0001   0003   C1 C2 C3-----...--
```

DSORG return specification - Key = '0057'

DALRTORG requests that the data set organization of the allocated data set be returned to the caller of DYNALLOC When you code DALRTORG, # must be one, LEN must be two, and PARM is a two-byte field. Dynamic allocation puts one of the following values into PARM. Note that when the requested information is not available, the system returns a value of X'0000' in PARM rather than a LEN of 0.

X'0000'
Dynamic allocation cannot determine the DSORG

X'0004'
TR

X'0008'
VSAM

X'0020'
TQ

X'0040'
TX

X'0080'
GS

X'0200'
PO

X'0300'
POU

X'0400'
MQ

X'0800'
CQ

X'1000'
CX

X'2000'
DA

X'2100'
DAU

X'4000'
PS

X'4100'
PSU

X'8000'
IS

X'8100'
ISU

The system does not fail the dynamic allocation request if it cannot obtain the data set organization (DSORG). Therefore, when you use this key, provide an SVC 99 request block extension (see "Setting up the request block extension" on page 555). Upon return, verify that fields S99EERR and S99EINFO in the request block extension are zero. If these fields are non-zero, do not use the returned DSORG.

Example: To specify that the DSORG be returned, code:

```
KEY    #      LEN    PARM
0057   0001   0002   --
```

This specification would be updated for a DSORG of PS as follows:

```
KEY    #      LEN    PARM
0057   0001   0002   4000
```

Subsystem request specification - key = '005C'

DALSSREQ requests that a subsystem data set be allocated and, optionally, specifies the name of the subsystem for which the data set is to be allocated.

When you code DALSSREQ without specifying a subsystem name, # must be zero and LEN and PARM are not specified. The data set is then allocated to the primary subsystem.

When you code the subsystem name in the DALSSREQ key, # must be one, LEN is the length of the subsystem name, up to a maximum of 4, and PARM contains the subsystem name.

Note: To specify DALSSREQ, your program must be APF-authorized, in supervisor state, or running with PSW key 0 - 7. If your program does not meet these requirements, you may be able to use the DALUASSR text unit instead.

Example 1: To request a subsystem data set for the primary subsystem, code:

```
KEY    #      LEN   PARM
005C   0000   -     -
```

Example 2: To request a subsystem data set for JES2, code:

```
KEY    #      LEN    PARM
005C   0001   0004   D1 C5 E2 F2
```

Volume serial return specification - Key = '005D'

DALRTVOL requests that the volume serial number associated with the allocated data set be returned. Only the first volume serial of a multiple-volume data set is returned, and the volume sequence number, if any, is ignored.

When you code DALRTVOL, # must be one, LEN must be six, and PARM is a six-byte field.

If the allocated volume serial is available at the completion of allocation, dynamic allocation puts the number in PARM. If the volume serial is not available at the completion of allocation, dynamic allocation sets LEN to zero.

The volume serial will not be available at the completion of allocation if either of the following is true:

- No volume serial is allocated to the data set (a VIO or job entry subsystem data set)
- The request results in the allocation of a new data set on magnetic tape without a specific volume serial having been assigned.

The system does not fail the dynamic allocation request if it cannot obtain the volume serial number. Therefore, when you use this key, provide an SVC 99 request block extension (see "Setting up the request block extension" on page 555). Upon return, verify that fields S99EERR and S99EINFO in the request block extension are zero. If these fields are non-zero, do not use the returned volume serial number.

Example: To specify that the allocated volume serial be returned, code:

```
KEY    #      LEN    PARM
005D   0001   0006   ------
```

This specification would be updated for the allocation of data set ABC on volume 123456 as follows:

```
KEY           LEN    PARM
005D   0001   0006   F1 F2 F3 F4 F5 F6
```

Subsystem request type specification - Key = '0062'

DALSSATT specifies that the subsystem data set being requested is to be allocated to SYSIN. The subsystem request key (DALSSREQ) is required with this key.

Note: To specify DALSSATT, you must be APF-authorized, or in supervisor state, or running with PSW key 0 - 7.

When you code DALSSATT, # and LEN must be one, and PARM contains '80', for SYSIN data set.

Example: To specify a subsystem SYSIN data set, code:

```
KEY    #      LEN    PARM
0062   0001   0001   80
```

Spool data set browse token specification - Key = '006E'

DALBRTKN specifies a spool data set browse token that contains information about a JES2 spool data set that a user asks to browse. In addition to parameters about the spool data set, the token also contains parameters that you can use with the security authorization facility (SAF) to check the user's browse authorization.

When you code DALBRTKN, # must be 7. Mapping macro IAZBTOKP maps the length and parameter portion of this text unit. For a description of the IAZBTOKP mapping macro, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Example: Define the spool data set browse token in IAZBTOKP as follows:

- BTOK is the name of the spool data set browse token (BTOKID).
- 1 is the version number of the parameter list for BTOK (BTOKVERNM).

- 01001302 is the I/O table (IOT) module track record (MTTR) pointer (BTOKIOTP).
- A9BC2033 is the spool data set job key (BTOKJKEY).
- 017E is the ASID of the job that owns the spool data set (BTOKASID).
- IBMUSER is the RECVR userid you can use on a SAF call to check the authority of the browse request (BTOKRCID).
- DATA SET BROWSE is the LOGSTR data associated with IBMUSER (BTOKLOG5).

```
KEY   #    LEN1      PARM1  LEN2      PARM2  ... LEN7     PARM7
006E  007  IAZBTOKP  DSECT  containing the values associated with BTOK
```

Using the IAZBTOKP field names, the result is:

```
KEY   #     BTOKPL1  BTOKID    BTOKPL2  BTOKVRNM  BTOKPL3  BTOKIOTP
006E  0007  0004     C2E3D6D2  0002     0001      0004     01001302

            BTOKPL4  BTOKJKEY  BTOKPL5  BTOKASID  BTOKPL6  BTOKRCID
            0004     A9BC2033  0002     017E      0008     C9C2D4E4 C2C5D940

            BTOKPL7  BTOKLOGS
            00FF     0FC4C1E3  C140E2C5 C340C2D9  D6E6E2C5
```

Volume interchange specification - Key = '006F'

DALINCHG specifies the media type and track recording technique required for system-managed tape library allocation.

Whenever possible, IBM suggests that you use an installation-defined DATACLAS construct name to control cartridge media type and track recording technique. Use this key only when it is not possible to use a pre-defined DATACLAS construct because of the dynamic nature of the program and because the program must control the media type and track recording technique. Contact your storage administrator before using this key.

Note: To specify DALINCHG, your program must be APF-authorized, in supervisor state, or running in PSW key 0-7. The specification of DALINCHG will be ignored if a non-system-managed tape volume is allocated.

When you code this key, # must be one, but LEN and PARM can be either 1 byte or 2 bytes in length. PARM must contain one of the following values:

01
0001
Cartridge System Tape requested; no specific track recording technique requested.

11
0101
Cartridge System Tape requested; 18-track recording technique requested.

21
0201
Cartridge System Tape requested; 36-track recording technique requested.

02
0002
Enhanced Capacity Cartridge System Tape requested; no specific track recording technique requested.

22

0202
: Enhanced Capacity Cartridge System Tape requested; 36-track recording technique requested.

03
0003
: High Performance Tape Cartridge requested; no specific track recording technique requested.

33
0303
: High Performance Tape Cartridge requested; 128-track recording technique requested.

43
0403
: High Performance Tape Cartridge requested; 256-track recording technique requested.

53
0503
: High Performance Tape Cartridge requested; 384-track recording technique requested.

04
0004
: Extended High Performance Tape Cartridge requested; no specific track recording technique requested.

34
0304
: Extended High Performance Tape Cartridge requested; 128-track recording technique requested.

44
0404
: Extended High Performance Tape Cartridge requested; 256-track recording technique requested.

54
0504
: Extended High Performance Tape Cartridge requested; 384-track recording technique requested.

05
0005
: Enterprise Tape Cartridge 3592 requested; no specific recording technology requested.

65
0605
: Enterprise Tape Cartridge 3592 requested; Enterprise Format 1 recording technology requested.

75
0705
: Enterprise Tape Cartridge 3592 requested; Enterprise Format 2 recording technology requested.

85

0805
Enterprise Tape Cartridge 3592 requested; Enterprise Encrypted Format 2 recording technology requested.

95
0905
Enterprise Tape Cartridge 3592 requested; Enterprise Format 3 recording technology requested.

A5
0A05
Enterprise Tape Cartridge 3592 requested; Enterprise Encrypted Format 3 recording technology requested.

06
0006
Enterprise WORM Tape Cartridge 3592 requested; no specific recording technology requested.

66
0606
Enterprise WORM Tape Cartridge 3592 requested; Enterprise Format 1 recording technology requested.

76
0706
Enterprise WORM Tape Cartridge 3592 requested; Enterprise Format 2 recording technology requested.

86
0806
Enterprise WORM Tape Cartridge 3592 requested; Enterprise Encrypted Format 2 recording technology requested.

96
0906
Enterprise WORM Tape Cartridge 3592 requested; Enterprise Format 3 recording technology requested.

A6
0A06
Enterprise WORM Tape Cartridge 3592 requested; Enterprise Encrypted Format 3 recording technology requested.

07
0007
Enterprise Economy Tape Cartridge 3592 requested; no specific recording technology requested.

67
0607
Enterprise Economy Tape Cartridge 3592 requested; Enterprise Format 1 recording technology requested.

77
0707
Enterprise Economy Tape Cartridge 3592 requested; Enterprise Format 2 recording technology requested.

87

0807
: Enterprise Economy Tape Cartridge 3592 requested: Enterprise Encrypted Format 2 recording technology requested.

97
0907
: Enterprise Economy Tape Cartridge 3592 requested; Enterprise Format 3 recording technology requested.

A7
0A07
: Enterprise Economy Tape Cartridge 3592 requested; Enterprise Encrypted Format 3 recording technology requested.

08
0008
: Enterprise Economy WORM Tape Cartridge 3592 requested; no specific recording technology requested.

68
0608
: Enterprise Economy WORM Tape Cartridge 3592 requested; Enterprise Format 1 recording technology requested.

78
0708
: Enterprise Economy WORM Tape Cartridge 3592 requested; Enterprise Format 2 recording technology requested.

88
0808
: Enterprise Economy WORM Tape Cartridge 3592 requested; Enterprise Encrypted Format 2 recording technology requested.

98
0908
: Enterprise Economy WORM Tape Cartridge 3592 requested; Enterprise Format 3 recording technology requested.

A8
0A08
: Enterprise Economy WORM Tape Cartridge 3592 requested; Enterprise Encrypted Format 3 recording technology requested.

09
0009
: Enterprise Extended Tape Cartridge 3592 requested; no specific recording technology requested.

79
0709
: Enterprise Extended Tape Cartridge 3592 requested; Enterprise Format 2 recording technology requested.

89
0809
: Enterprise Extended Tape Cartridge 3592 requested; Enterprise Encrypted Format 2 recording technology requested.

99

0909
Enterprise Extended Tape Cartridge 3592 requested; Enterprise Format 3 recording technology requested.

A9
0A09
Enterprise Extended Tape Cartridge 3592 requested; Enterprise Encrypted Format 3 recording technology requested.

B9
0B09
Enterprise Extended Tape Cartridge 3592 requested; Enterprise format 4 recording technology requested.

C9
0C09
Enterprise Extended Tape Cartridge 3592 requested; Encrypted Enterprise format 4 recording technology requested.

0A
000A
Enterprise Extended WORM Tape Cartridge 3592 requested; no specific recording technology requested.

7A
070A
Enterprise Extended WORM Tape Cartridge 3592 requested; Enterprise Format 2 recording technology requested.

8A
080A
Enterprise Extended WORM Tape Cartridge 3592 requested; Enterprise Encrypted Format 2 recording technology requested.

9A
090A
Enterprise Extended WORM Tape Cartridge 3592 requested; Enterprise Format 3 recording technology requested.

AA
0A0A
Enterprise Extended WORM Tape Cartridge 3592 requested; Enterprise Encrypted Format 3 recording technology requested.

BA
0B0A
Enterprise Extended WORM Tape Cartridge 3592 requested; Enterprise format 4 recording technology requested.

CA
0C0A
Enterprise Extended WORM Tape Cartridge 3592 requested; Encrypted Enterprise format 4 recording technology requested.

0B
000B
Enterprise Advanced Tape Cartridge 3592; no specific recording technology requested.

BB
0B0B
Enterprise Advanced Tape Cartridge 3592; Enterprise format 4 recording mode.

CB
0C0B
Enterprise Advanced Tape Cartridge 3592; Encrypted Enterprise format 4 recording mode.

0C
000C
Enterprise Advanced WORM Tape Cartridge 3592; no specific recording technology requested.

BC
0B0C
Enterprise Advanced WORM Tape Cartridge 3592; Enterprise format 4 recording mode.

CC
0C0C
Enterprise Advanced WORM Tape Cartridge 3592; Encrypted Enterprise format 4 recording mode.

0D
000D
Enterprise Advanced Economy Tape Cartridge 3592; no specific recording technology requested.

BD
0B0D
Enterprise Advanced Economy Tape Cartridge 3592; Enterprise format 4 recording mode.

CD
0C0D
Enterprise Advanced Economy Tape Cartridge 3592; Encrypted Enterprise format 4 recording mode.

Example: To request Cartridge System Tape recorded in 36-track format, code:

```
KEY    #      LEN    PARM
006F   0001   0001   21
```

To request the same thing with a 2 byte DALINCHG parameter, code:

```
KEY  #    LEN  PARM
006F 0001 0002 0201
```

Subsystem request specification - Key = '0075'

DALUASSR requests that a subsystem data set be allocated and, optionally, specifies the name of the subsystem for which the data set is to be allocated.

This request is similar to the DALSSREQ request, but can be used by unauthorized callers.

When you code DALUASSR without specifying a subsystem name, # must be zero and LEN and PARM are not specified. The data set is then allocated to the primary subsystem.

When you code the subsystem name in the DALUASSR key, # must be one, LEN is the length of the subsystem name, up to a maximum of 4, and PARM contains the subsystem name.

Example 1: To request a subsystem data set for the primary subsystem, code:

```
KEY   #     LEN      PARM
0075  0000  -        -
```

Example 2: To request a subsystem data set for JES2, code:

```
KEY   #     LEN      PARM
0075  0001  0004     D1 C5 E2 F2
```

Uniquely allocated temporary data set - Key = '0077'

DALUNQDS indicates that a temporary data set is being allocated and that the address space allocating the data set will only allocate the specified data set name, or generated data set name, to the DD currently being allocated, and no other DD.

Dynamic allocation normally tracks temporary data set names when they are allocated and uses this information to avoid deleting a temporary data set more than once when the data set is allocated multiple times. Use of this text unit indicates that the dynamic allocation caller will ensure that the temporary data set is only allocated once and that the system can avoid this processing.

When you code DALUNQDS, # must be zero. LEN and PARM are not specified.

Example: To indicate that a temporary data set is uniquely allocated:

```
KEY   #     LEN  PARM
0077  0000  -    -
```

Dynamic unallocation text units

Use verb code 02 and the text unit keys listed in Table 88 and described on the following pages to request dynamic unallocation processing by DYNALLOC. To deallocate a resource, you must specify either the DUNDDNAM key, the DUNDSNAM key, or the DUNPATH key.

Table 88. Verb Code 02 (Dynamic Unallocation) – Text Unit Keys, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0001	DUNDDNAM	Specifies the ddname of the resource to be deallocated.
0002	DUNDSNAM	Specifies the data set to be deallocated.
0003	DUNMEMBR	Specifies the PDS member to be deallocated.
0005	DUNOVDSP	Specifies an overriding disposition for the data set to be unallocated.
0007	DUNUNALC	Specifies deallocation even if the resource has the permanently allocated attribute.
0008	DUNREMOV	Specifies removal of the “in-use” attribute, even if the resource does not have the permanently allocated attribute.
000A	DUNOVSNH	Specifies “nohold” status for a deallocated SYSOUT data set and overrides previous “hold” specification.
0018	DUNOVCLS	Specifies an overriding SYSOUT class.
0058	DUNOVSUS	Specifies an overriding destination to which the SYSOUT data set being deallocated is to be routed. **Note:** Keys 0058 and 0063 are equivalent to the JCL **DEST=** parameter, where **DEST=(***dunovsus***)** or **DEST=(***dunovsus,dunovuid***)**.
0059	DUNOVSHQ	Specifies “hold” status for a deallocated SYSOUT data set and overrides previous “nohold” specifications.

Table 88. Verb Code 02 (Dynamic Unallocation) – Text Unit Keys, Mnemonics, and Functions (continued)

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0063	DUNOVUID	Specifies an overriding destination user ID to which the SYSOUT data set being deallocated is to be routed. **Note:** Keys 0058 and 0063 are equivalent to the JCL **DEST=** parameter, where **DEST=(***dunovsus***)** or **DEST=(***dunovsus,dunovuid***)**.
8013	DUNSPIN	Specifies an overriding decision that determines whether the output for the SYSOUT data set is to be printed immediately, or at the end of the job.
8017	DUNPATH	Specifies the pathname of the z/OS UNIX file to be deallocated.
801A	DUNOVPDS	Specifies an overriding disposition for the file to be deallocated.

Ddname specification - Key = '0001'

DUNDDNAM specifies the ddname of the resource to be deallocated. When you code this key, # must be one, LEN is the length of the ddname, up to a maximum of 8, and PARM contains the ddname.

Example: To specify the ddname DD1, code:

```
KEY    #      LEN    PARM
0001   0001   0003   C4 C4 F1
```

Dsname specification - Key = '0002'

DUNDSNAM specifies the data set name to be deallocated. The data set name can contain special characters if the data set name is enclosed in apostrophes. The dsname can contain system symbols. See the information on using system symbols in *z/OS MVS Initialization and Tuning Reference*. When you code this key, # must be one, LEN contains the length of the dsname, and PARM contains the dsname.

The maximum length of the data set name is 44 characters, excluding any enclosing apostrophes and compressing any double apostrophes within the data set name.

Example: To specify the dsname MYDATA, code:

```
KEY    #      LEN    PARM
0002   0001   0006   D4 E8 C4 C1 E3 C1
```

Member name specification - Key = '0003'

DUNMEMBR specifies that a particular member of the data set is to be deallocated. The member name can contain system symbols. See the information on using system symbols in *z/OS MVS Initialization and Tuning Reference*.

The dsname unallocation key (DUNDSNAM) is required with this key. When you code DUNMEMBR, # must be one, LEN is the length of the member name, up to a maximum of 8, and PARM contains the member name.

Example: To specify the member name MEM1, code:

```
KEY    #      LEN    PARM
0003   0001   0004   D4 C5 D4 F1
```

Overriding disposition specification - Key = '0005'

DUNOVDSP specifies a disposition that overrides the disposition assigned to a data set when it was allocated. When you code DUNOVDSP, # and LEN must be one, and PARM contains one of the following values:

X'01' An overriding disposition of UNCATLG

X'02' An overriding disposition of CATLG

X'04' An overriding disposition of DELETE

X'08' An overriding disposition of KEEP

Example: To specify an overriding disposition of CATLG, code:

```
KEY    #      LEN    PARM
0005   0001   0001   02
```

Note: Dynamic allocation ignores this key if any of the following are true:

- The overriding disposition was DELETE and the data set was originally allocated as SHARE.
- The data set was originally allocated with a disposition of PASS.
- The data set is a VSAM data set and SMS is not active on the system.
- The data set is a non-subsystem data set that has a system-generated name.

When dynamic allocation must ignore a DUNOVDSP request, it still performs the deallocation processing, but uses the disposition from the original allocation request.

Unallocate option specification - Key = '0007'

DUNUNALC specifies that the resource is to be deallocated even if it has the permanently allocated attribute. The remove in-use option key (DUNREMOV) is mutually exclusive with DUNUNALC. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify the unalloc option, code:

```
KEY    #      LEN   PARM
0007   0000   -     -
```

Remove in-use option specification - Key = '0008'

DUNREMOV specifies that the in-use attribute is to be removed even if the resource does not have the permanently allocated attribute. The unalloc option key (DUNUNALC) is mutually exclusive with DUNREMOV. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify the remove option, code:

```
KEY    #      LEN   PARM
0008   0000   -     -
```

Overriding SYSOUT nohold specification - Key = '000A'

DUNOVSNH specifies that the SYSOUT data set being deallocated is not to be placed on the hold queue. This specification overrides the HOLD/NOHOLD specification assigned when the data set was allocated.

This key is ignored if the data set is not a SYSOUT data set or if the data set is the internal reader. The overriding hold key (DUNOVSHQ) is mutually exclusive with DUNOVSNH. When you code this key. # must be zero; LEN and PARM are not specified.

Example: To specify nohold, code:

```
KEY    #      LEN   PARM
000A   0000   -     -
```

Overriding SYSOUT class specification - Key = '0018'

DUNOVCLS specifies a SYSOUT class that overrides the class assigned when the SYSOUT data set was allocated. This key is ignored if the resource is not a SYSOUT data set. When you code DUNOVCLS, # and LEN must be one, and PARM contains the overriding class.

The SYSOUT class characteristics are used in processing the output, with the exception of the spool space allocation attribute. The spool space allocation for the SYSOUT is unchanged from what was specified at data set allocation time, either through the SYSOUT class definition in JES or through dynamic allocation.

Example: To specify an overriding class of C, code:

```
KEY    #      LEN    PARM
0018   0001   0001   C3
```

Overriding SYSOUT remote workstation specification - Key = '0058'

In a JES2 environment, DUNOVSUS specifies that, upon deallocation, the SYSOUT data set being deallocated is to be routed to a remote workstation, node, special local, or userid. In a JES3 environment, DUNOVSUS specifies that, upon deallocation, the SYSOUT data set being deallocated is to be routed to ANYLOCAL, JES3-defined device name, device number, JES3 defined symbolic group name, or NJE node. The system ignores DUNOVSUS if the data set is not a SYSOUT data set.

When you code DUNOVSUS, # must be one, LEN is the length of the destination name specified in PARM, up to a maximum of 8, and PARM contains the remote workstation node, special local, userid, ANYLOCAL, device name, device number, symbolic group name, or NJE node.

Example: To specify the remote work station USER01, code:

```
KEY    #      LEN    PARM
0058   0001   0006   E4 E2 C5 D9 F0 F1
```

Overriding SYSOUT hold queue specification - Key = '0059'

DUNOVSHQ specifies that the SYSOUT data set being deallocated, if is a HOLD class will be placed on the HOLD queue,, otherwise it will be placed in HOLD status on the WRITER queue. This specification overrides the HOLD/NOHOLD specification assigned when the data set was allocated. This key is ignored if the data set is not a SYSOUT data set or if the data set is the internal reader. The overriding nohold key (DUNOVSNH) is mutually exclusive with this key.

When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify hold, code:

```
KEY    #       LEN   PARM
0059   0000    -     -
```

Overriding SYSOUT user ID specification - Key = '0063'

DUNOVUID specifies that, upon deallocation, the SYSOUT data set being deallocated is to be routed to the specified user ID, remote device, or special local. Specifying DUNOVUID overrides the user ID, remote device, or special local specification assigned when the data set was allocated. The system ignores DUNOVUID in a JES3 environment, or if you specify it for a data set that is not a SYSOUT data set.

If you specify DUNOVUID, you must also specify DUNOVSUS.

When you code DUNOVUID, # must be one, LEN is the length of the destination specified in PARM, up to a maximum of 8, and PARM contains the user ID, remote device, or special local. The user ID can contain any EBCDIC characters, including special characters.

Example: To specify that, upon unallocation, the SYSOUT data set is to be routed to user ID D58-VWM at node USER01, code:

```
KEY    #     LEN   PARM
0063   0001  0007  C4 F5 F8 60 E5 E6 D4
0058   0001  0006  E4 E2 C5 D9 F0 F1
```

Overriding spin data set specification - Key = '8013'

DUNSPIN specifies whether the output for the SYSOUT data set is to be printed immediately or at the end of the job. This specification overrides the SPIN value assigned when the SYSOUT data set was allocated.

When you code DUNSPIN, both # and LEN must be 1; PARM must contain one of the following:

X'80' Data set available for printing when it is unallocated

X'40' Data set available for printing at the end of the job.

Example: To specify that the output be printed when the data set is unallocated, code:

```
KEY    #     LEN   PARM
8013   0001  0001  80
```

z/OS UNIX file path specification - Key = '8017'

DUNPATH specifies the path specification for the z/OS UNIX file to be deallocated. The pathname can contain system symbols. See the information on using system symbols in *z/OS MVS Initialization and Tuning Reference*.

When you code DUNPATH, # must be 1 and LEN is the length of the z/OS UNIX file pathname; PARM contains the z/OS UNIX file pathname. The maximum length of the pathname is 255 characters.

Example: To specify the z/OS UNIX file pathname **/u/myuid/myapp/scr.dat**, code:

```
KEY    #     LEN   PARM
8017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
```

z/OS UNIX overriding file disposition specification - Key = '801A'

DUNOVPDS specifies the file disposition of the z/OS UNIX file being deallocated. It overrides the disposition assigned to a file when it was allocated.

When you code DUNOVPDS, # must be 1 and LEN must be 1; PARM must contain one of the following values:

- X'04' - the file is to be deleted
- X'08' - the file is to be kept

Example: To specify that the z/OS UNIX file being deallocated is deleted, code:

```
KEY    #     LEN   PARM
801A   0001  0001  04
```

Dynamic concatenation text units

Use verb code 03 and the text units listed in Table 89 and described in the following paragraphs to request dynamic concatenation processing by DYNALLOC. You must specify DCCDDNAM to dynamically concatenate a resource.

Table 89. Verb Code 03 (Dynamic Concatenation) – Text Unit Keys, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0001	DCCDDNAM	Specifies the ddnames to be concatenated.
0004	DCCPERMC	Specifies the permanently concatenated attribute.

Ddname specification - Key = '0001'

DCCDDNAM specifies the ddnames that are associated with the data sets to be concatenated. When you code DCCDDNAM, # is the number of ddnames being specified, which must be at least 2 and can be up to a maximum of 1635 (X'663'). LEN is the length of the immediately following ddname, up to a maximum of 8, and PARM contains the ddname.

Example: To specify concatenation of SYSLIB to MYLIB, code:

```
KEY    #      LEN    PARM         LEN    PARM
0001   0002   0005   D4E8D3C9C2   0006   E2E8E2D3C9C2
```

Permanently concatenated attribute specification - Key = '0004'

DCCPERMC specifies that the concatenated group be assigned the permanently concatenated attribute. A permanently concatenated group cannot be deconcatenated. When you code this key, # must be zero; LEN and PARM are not specified.

Example: To specify assignment of the permanently concatenated attribute, code:

```
KEY    #      LEN   PARM
0004   0000   -     -
```

Dynamic deconcatenation text unit

Use verb code 04 and DDCDDNAM to request dynamic deconcatenation processing by DYNALLOC. You cannot deconcatenate a permanently concatenated data set.

Table 90. Verb Code 04 (Dynamic Deconcatenation) – Text Unit Key, Mnemonic, and Function

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0001	DDCDDNAM	Specifies the ddname of the group to be deconcatenated.

Ddname specification - Key = '0001'

DDCDDNAM specifies the ddname of the concatenated group that is to be deconcatenated. DDCDDNAM is required for dynamic deconcatenation.

When you code DDCDDNAM, # must be one, LEN is the length of the ddname, up to a maximum of 8, and PARM contains the ddname.

Example: To request the deconcatenation of the group of data sets associated with the ddname DD1, code:

```
KEY    #      LEN    PARM
0001   0001   0003   C4 C4 F1
```

Text units for removing the in-use attribute based on task ID

Use verb code 05 and the text units in Table 91 and described as follows to request that dynamic allocation remove the in-use attribute from resources based on task-ID. You must specify either the DRITCBAD key or the DRICURNT key to remove the in-use attribute. You cannot specify both keys for a single remove in-use request.

Table 91. Verb Code 05 (Remove In-Use Processing Based on Task-ID) – Text Unit Keys, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0001	DRITCBAD	Removes the “in-use” attribute from all resources associated with the specified TCB address.
0002	DRICURNT	Removes the “in-use” attribute from all resources in the address space associated with tasks that have terminated.

TCB address specification - Key = '0001'

DRITCBAD specifies that the in-use attribute is to be removed from all resources associated with the specified TCB address. The current task option key (DRICURNT) is mutually exclusive with this key.

When you code DRITCBAD, # must be one, LEN must be four, and PARM contains the TCB address.

Example: To specify the TCB address 22AC0, code:

```
KEY    #      LEN    PARM
0001   0001   0004   00022AC0
```

Current task option specification - Key = '0002'

DRICURNT specifies that the in-use attribute is to be removed from all resources in the address space associated with tasks that have terminated. This key is mutually exclusive with the TCB address key (DRITCBAD). When you code DRICURNT, # must be zero; LEN and PARM are not specified.

Example: To specify the current task option, code:

```
KEY    #      LEN   PARM
0002   0000   -     -
```

Ddname allocation text units

Use verb code 06 and the text units listed in Table 92 on page 697 and described as follows to request ddname allocation processing. In ddname allocation, you are specifying that dynamic allocation is to use a particular existing allocation to satisfy your allocation request. You identify the data set you wish by specifying the ddname associated with it.

See “Using ddname allocation” on page 540 for requirements and restrictions when allocating resources by ddname.

Table 92. Verb Code 06 (Ddname Allocation) – Text Unit Keys, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	DYNALLOC Function
0001	DDNDDNAM	Specifies the ddname to be allocated.
0002	DDNRTDUM	Requests a dummy data set indication.

Ddname specification - Key = '0001'

DDNDDNAM specifies the ddname of the resource to be allocated. It is required for dynamic allocation by ddname.

When you code DDNDDNAM, # must be one, LEN contains the length of the ddname, up to a maximum of 8, and PARM contains the ddname.

Example: To specify the ddname SYSLIB, code:

```
KEY    #      LEN    PARM
0001   0001   0006   E2 E8 E2 D3 C9 C2
```

Return DUMMY indication specification - Key = '0002'

Code DDNRTDUM to request the return of an indication if the ddname specified in DDNDDNAM is associated with a dummy data set. When you code DDNRTDUM, # and LEN must be one, and PARM is a one-byte field. DYNALLOC sets PARM as follows:

X'80' The ddname is associated with a dummy data set

X'00' Otherwise

Example: To specify that the DUMMY indication be returned, code:

```
KEY    #      LEN    PARM
0002   0001   0001   -
```

Dynamic information retrieval text units

Use verb code 07 and the text units listed in Table 93 on page 698 and described as follows to request that dynamic allocation return certain information about the allocated resources. You must specify the DINDDNAM key, the DINDSNAM key, the DINPATH key, or the DINRELNO key to request information retrieval. You cannot specify more than one of these keys for a single information retrieval request.

Keys X'0001', X'0002', and X'000F', and X'8017' are input keys. You use them to supply information to the system. The remaining keys are output keys, which the system uses to return the information you requested.

If you request that the system return information that does not exist, the system sets the length field for that parameter to zero, and there will be no data field for that parameter. The few exceptions to this rule are noted in the individual text unit descriptions.

For example, if you request information about the storage class of a data set, but that data set is not SMS-managed, the system returns zero in the length field of the text unit.

For either verb code 01 or verb code 07 information retrieval, the data returned in the PARM field is valid only for the number of bytes returned in the LEN field. If on input the PARM field is longer than the information returned on output, the remaining bytes in PARM might contain data from previous invocations of DYNALLOC. The information retrieval function does not pad the remaining bytes with blanks.

For example, on **Input** the text unit initializes the PARM field to “AAAAAAAA”. On **Output** the text unit contains the name “MYDD” and a LEN of X'0004'. Beyond the valid length (4 bytes) of “MYDD”, the PARM field might contain data from the previous request (“MYDDAAAA”).

```
        KEY  #     LEN    PARM
Input:  0004 0001  0008   C1C1C1C1C1C1C1C1

Output: 0004 0001  0004   D4E8C4C4C1C1C1C1
```

When you request information for multiple parameters, the LEN and PARM fields of the next parameter immediately follow the information returned from the previous parameter. For example, the **Output** text unit below contains a second LEN and PARM pair. The request returns the information “X” in the PARM2 field. The information in the fields is **contiguous**; spaces are shown in the example only for readability.

```
KEY  #     LEN1   PARM1     LEN2     PARM2
Output: nnnn 0002  0004   D4E8C4C4  0001     E7
```

Table 93. Verb Code 07 (Dynamic Information Retrieval) – Text Unit Keys, Mnemonics, and Functions

Hex Text Unit Key	Mnemonic	DYNALLOC Function (Input)
0001	DINDDNAM	Specifies the ddname identifier of the requested information.
0002	DINDSNAM	Specifies the data set for which the information is requested.
0004	DINRTDDN	Requests the associated ddname.
0005	DINRTDSN	Requests the data set name.
0006	DINRTMEM	Requests the PDS member name.
0007	DINRTSTA	Requests the data set's status.
0008	DINRTNDP	Requests the data set's normal disposition.
0009	DINRTCDP	Requests of the data set's conditional disposition.
000A	DINRTORG	Requests the data set's organization.
000B	DINRTLIM	Requests the number of resources that must be deallocated before making a new allocation.
000C	DINRTATT	Requests the special attribute indications.
000D	DINRTLST	Requests a last relative entry indication.
000E	DINRTTYP	Requests the data set's type (terminal or dummy).
000F	DINRELNO	Specifies the desired allocation information retrieval by relative request number.
0010	DINRTVOL	Requests the return of the first volume serial number associated with the specified allocation.
0011	DINRTDDX	Requests the return of the ddname associated with the specified allocation.
0012	DINRLPOS	Requests the return of the relative position of a data set within a concatenated group.
8017	DINPATH	Specifies the pathname of the z/OS UNIX file for which information is requested.
C003	DINRCNTL	Requests the JCL CNTL statement reference.

Table 93. Verb Code 07 (Dynamic Information Retrieval) – Text Unit Keys, Mnemonics, and Functions (continued)

Hex Text Unit Key	Mnemonic	DYNALLOC Function (Input)
C004	DINRSTCL	Requests the storage class of a new SMS-managed data set.
C005	DINRMGCL	Requests the management class of a new SMS-managed data set.
C006	DINRDACL	Requests the data class of a new data set.
C00B	DINRRECO	Requests the organization of a new VSAM data set.
C00C	DINRKEYO	Requests the key offset of a new VSAM data set.
C00D	DINRREFD	Requests the DD name specified by the REFDD parameter of the DD statement.
C00E	DINRSECM	Requests the name of the RACF security data set profile.
C00F	DINRLIKE	Requests the data set name on the LIKE parameter.
C010	DINRAVGR	Requests the value of the unit of allocation for a data set.
C012	DINRDSNT	Requests the data set type, such as PDS.
C013	DINRSPIN	Requests the spin data set specification.
C014	DINRSEGM	Requests the number of logical, line-mode pages (the segment) to be produced for a SYSOUT data set before the segment becomes eligible for immediate printing.
C017	DINRPATH	Requests the return of the z/OS UNIX file path specification.
C018	DINRPOPT	Requests the return of the z/OS UNIX file options.
C019	DINRPMDE	Requests the return of z/OS UNIX file access attributes.
C01A	DINRPNDS	Requests the return of the disposition of the z/OS UNIX file during normal step termination.
C01B	DINRPCDS	Requests the return of the disposition of the z/OS UNIX file during abnormal step termination.
C01D	DINRFDAT	Requests the return of the organization of a z/OS UNIX file.
C02A	DINRSPI2	Requests the SPIN interval specification.
C02B	DINRSYML	Requests the SYMLIST specification.
C02C	DINRDSNV	Requests the DSNTYPE version.
C02D	DINRMAXG	When you have applied APAR OA42358, requests the value of MAXGENS.
C02E	DINRGDGO	Requests the return of the order in which the individual generation data sets (GDSs) are concatenated.

Ddname specification - Key = '0001'

DINDDNAM specifies the ddname associated with the allocation you are requesting information about. It is mutually exclusive with the dsname (DINDSNAM), relative entry (DINRELNO), and pathname (DINPATH) keys. When you code DINDDNAM, # must be one, LEN is the length of the ddname, and PARM contains the ddname.

Example: To request the return of the z/OS UNIX file pathname that is allocated to the ddname DD1, code:

```
KEY    #      LEN    PARM
0001   0001   0003   C4 C4 F1
C017   0001   0016   00...00...00...00...
```

Dsname specification - Key = '0002'

DINDSNAM specifies the dsname of the allocated resource about which you are requesting information. The data set name can contain special characters if the data set name is enclosed in apostrophes. The dsname can contain system symbols. See the topic on using system symbols and system symbols in *z/OS MVS JCL Reference* for more information. If the data set you specify is allocated to more than one ddname, Dynamic Information Retrieval will retrieve information based on the first ddname allocated to the specified dsname. It is mutually exclusive with the ddname (DINDDNAM), relative entry (DINRELNO), and pathname (DINPATH) keys. When you code DINDSNAM, # must be one, LEN is the length of the dsname, and PARM contains the dsname.

The maximum length of the data set name is 44 characters, excluding any enclosing apostrophes and compressing any double apostrophes within the data set name.

Example: To specify the dsname MYDATA, code:

```
KEY    #      LEN    PARM
0002   0001   0006   D4 E8 C4 C1 E3 C1
```

Return ddname specification - Key = '0004'

Code DINRTDDN to request the return of the ddname associated with the specified allocation. If the data set you specify is a member of a concatenated group, and is not the first member, there is no ddname associated with it. If you need to retrieve information for a data set that is a member of a concatenated group, consider using the DINRTDDX text unit instead.

When you code this key, # must be one, LEN must be at least the length of the ddname, and can be longer, up to a maximum of 8. The PARM field must be the length specified by the value in LEN. Upon return to your program, PARM will contain the requested ddname, and LEN will be set to its length.

Example: To request the return of the ddname, code:

```
KEY    #      LEN    PARM
0004   0001   0008   --------
```

Return dsname specification - Key = '0005'

Code DINRTDSN to request the return of the dsname of the specified allocation. When you code this key, # must be one, LEN must be at least the length of the dsname, and can be longer, up to a maximum of 255 characters. The PARM field must be the length specified by the value in LEN. Upon return to your program, PARM will contain the dsname and LEN will be set to its length.

Example: To request that the dsname be returned, code:

```
KEY    #      LEN    PARM
0005   0001   002C   --------
```

Return member name specification - Key = '0006'

Code DINRTMEM to request the return of the member name associated with the specified allocation. When you code this key, # must be one, LEN must be at least the length of the member name, and can be longer, up to a maximum of 255 characters. The PARM field must be the length specified by the value in LEN.

Upon return to your program, PARM will contain the member name and LEN will be set to its length (or to zero, if no member name is associated with the allocation.)

Example: To request that the member name be returned, code:

```
KEY    #      LEN    PARM
0006   0001   0008   --------
```

Return status specification - Key = '0007'

Code DINRTSTA to request the return of the data set status of the specified allocation. When you code this key, # and LEN must be one, and PARM is a one-byte field. Upon return to your program, the PARM field will contain one of the following values:

X'01' OLD

X'02' MOD

X'04' NEW

X'08' SHR

Example: To request that the status be returned, code:

```
KEY    #      LEN    PARM
0007   0001   0001   -
```

Return normal disposition specification - Key = '0008'

Code DINRTNDP to request the return of the normal disposition of the specified resource. When you code this key, # and LEN must be one. PARM is a one-byte field. Upon return to your program, PARM will contain one of the following values:

X'01' UNCATLG

X'02' CATLG

X'04' DELETE

X'08' KEEP

X'10' PASS

Example: To request that the normal disposition be returned, code:

```
KEY    #      LEN    PARM
0008   0001   0001   -
```

Return conditional disposition specification - Key = '0009'

Code DINRTCDP to request the return of the conditional disposition of the specified resource. When you code this key, # and LEN must be one. PARM is a one-byte field. Upon return to your program, PARM will contain one of the following values. Note that the system does not return 0 in the LEN field when the requested information is not defined.

X'00' Conditional disposition not defined

X'01' UNCATLG

X'02' CATLG

X'04' DELETE

X'08' KEEP

X'10' PASS

Example: To request that the conditional disposition be returned, code:

```
KEY    #      LEN    PARM
0009   0001   0001   -
```

Return data set organization specification key = '000A'

Code DINRTORG to request the return of the data set organization (DSORG) of the specified resource. When you code this key, # must be one, LEN must be two, and PARM is a two-byte field. Upon return to your program, PARM will contain one of the following. Note that the system does not return 0 in the LEN field when the requested information is not defined.

X'0000'
Dynamic allocation cannot determine the DSORG

X'0004'
TR

X'0008'
VSAM

X'0020'
TQ

X'0040'
TX

X'0080'
GS

X'0200'
PO

X'0300'
POU

X'0400'
MQ

X'0800'
CQ

X'1000'
CX

X'2000'
DA

X'2100'
DAU

X'4000'
PS

X'4100'
PSU

X'8000'
IS

X'8100'
ISU

Example: To request that the data set organization be returned, code:

```
KEY    #      LEN    PARM
000A   0001   0002   --
```

Return limit specification - Key = '000B'

Code DINRTLIM to request the return of the number of permanently allocated resources that must be deallocated before a new allocation can be made when the control limit has been exceeded. See “Control limit” on page 539 and “Permanently allocated attribute” on page 540 for further explanation.

When you code this key, # must be one, LEN must be two, and PARM is a two-byte field. Upon return to your program, PARM is set to the number of resources that need to be deallocated. Note that the system does not return 0 in the LEN field when the requested information is not defined.

Example: To request that the number of not-in-use data sets over the control limit be returned, code:

```
KEY    #      LEN    PARM
000B   0001   0002   --
```

If three data sets must be deallocated, dynamic allocation returns DINRTLIM as follows:

```
KEY    #      LEN    PARM
000B   0001   0002   0003
```

Return dynamic allocation attribute specification - Key = '000C'

Code DINRTATT to request indications of the attributes assigned to the specified resource. When you code this key, # and LEN must be one, and PARM is a one-byte field. Upon return to your program, PARM is set as follows:

Bit 0 On, if permanently concatenated

Bit 1 On, if in use

Bit 2 On, if permanently allocated

Bit 3 On, if convertible

Bit 4 On, if dynamically allocated

Bits 5-7
Not an intended programming interface.

Example: To request return of the data set attributes, code:

```
KEY    #      LEN    PARM
000C   0001   0001   --
```

If the allocation has the in-use and permanently allocated attributes, PARM contains the following on return:

```
KEY    #      LEN    PARM
000C   0001   0001   60
```

Return last entry specification - Key = '000D'

Code DINRTLST to determine if the relative entry request number, ddname, or dsname you specify is the last relative entry. When you code DINRTLST, # and LEN must be one, and PARM is a one-byte field. Upon return to your program, PARM contains one of the following values:

X'80' Last relative entry

X'00' Otherwise

Example: To request the return of the last entry indicator, code:

```
KEY    #      LEN    PARM
000D   0001   0001   -
```

Return data set type specification - Key = '000E'

Code DINRTTYP to determine the type of the specified data set. When you code this key, # and LEN must be one, and PARM is a one-byte field. Upon return to your program, PARM contains one of the following values:

X'80'	DUMMY data set
X'40'	Terminal allocation
X'20'	SYSIN data set
X'10'	SYSOUT data set
X'00'	otherwise

Example: To request return of the specified data set type, code:

```
KEY    #      LEN    PARM
000E   0001   0001   -
```

Relative request number specification - Key = '000F'

DINRELNO specifies the relative request number of the allocation you are requesting information about. It is mutually exclusive with the ddname (DINDDNAM), dsname (DINDSNAM), and pathname (DINPATH) keys. When you code DINRELNO, # must be one, LEN must be two, and PARM contains the relative number.

Example: To specify that information is to be returned about your tenth DYNALLOC request, code:

```
KEY    #      LEN    PARM
000F   0001   0002   000A
```

Return first volume serial specification - Key = '0010'

Code DINRTVOL to request the return of the first volume serial number associated with a specified allocated data set. When you code DINRTVOL, # must be one, LEN must be at least the length of the volume serial number but no more than 6. The PARM field will be the length specified by the value in LEN. Upon return to your program, PARM will contain the requested volume serial number and LEN will be set to its length.

Example: To request the return of the first volume serial number code:

```
KEY    #      LEN    PARM
0010   0001   0006   --------
```

Return ddname extended specification - Key = '0011'

Code DINRTDDX to request the return of the ddname associated with the specified allocation. If the data set you specify is a member of a concatenated group, this key returns the ddname associated with the first data set in the concatenation. Otherwise, this key returns the same information as the DINRTDDN text unit.

When you code this key, # must be one, LEN must be at least the length of the ddname, and can be longer, up to a maximum of 8. The PARM field must be the length specified by the value in LEN. Upon return to your program, PARM will contain the requested ddname, and LEN will be set to its length.

Example: To request the return of the ddname, code:

```
KEY    #      LEN    PARM
0011   0001   0008   --------
```

Return relative position specification - Key = '0012'

Code DINRLPOS to request the return of the relative position of a data set within a concatenated group. The first data set within a concatenated group has a relative position of zero. Likewise, if the data set you specify is not a member of a concatenated group, zero is returned.

This text unit can be used in combination with the DINRTDDX text unit to determine the ddname and relative position of a data set within a concatenated group.

When you code this key, # must be one, LEN must be 2, and PARM is a two-byte field. Upon return to your program, PARM will contain the relative position of the data set.

Example: To request the return of the relative position within a concatenated group, code:

```
KEY    #      LEN    PARM
0012   0001   0002   ----
```

z/OS UNIX file path specification - Key = '8017'

DINPATH specifies the pathname of the z/OS UNIX file for which you are requesting information. It is mutually exclusive with the ddname (DINDDNAM), dsname (DINDSNAM), and relative entry (DINRELNO) keys. The pathname can contain system symbols. See the information on using system symbols and system symbols in *z/OS MVS JCL Reference* for more information.

When you code DINPATH, # must be 1, LEN is the length of the pathname, and PARM contains the pathname.

Example: To specify the z/OS UNIX file pathname **/u/myuid/myapp/scr.dat**, code:

```
KEY    #     LEN   PARM
8017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
```

CNTL specification - Key = 'C003'

DINRCNTL requests the name of the JCL CNTL statement being referenced. When you code DINRCNTL, # must be one, LEN must be at least 26 and can be longer, and the PARM field is the length specified by the value in LEN.

If a CNTL statement reference exists, the system returns the name into PARM. If the name is fewer than 26 bytes long, it is left-justified and padded to the right with blanks. The system also returns 26 into LEN.

Example: To request the label of the referenced JCL CNTL statement, code:

```
KEY    #      LEN    PARM
C003   0001   001A   - - - ...
```

Return storage class specification - Key = 'C004'

DINRSTCL requests the storage class of the specified SMS-managed data set. When you code DINRSTCL, # must be 1, LEN must be at least 8 and can be longer, and the PARM field must be the length specified by the value in LEN.

If a storage class is associated with the data set, the system returns the storage class identifier into PARM. If the identifier is fewer than 8 bytes long, it is left-justified and padded to the right with blanks. The system also returns 8 into LEN.

Example: To request the storage class of the specified SMS-managed data set, code:

```
KEY    #      LEN    PARM
C004   0001   0008   - - - - - - - -
```

Return management class specification - Key = 'C005'

DINRMGCL requests the management class of the specified SMS-managed data set. When you code DINRMGCL, # must be 1, LEN must be at least 8 and can be longer, and the PARM field must be the length specified by the value in LEN.

If a management class is associated with the data set, the system returns the management class identifier into PARM. If the identifier is fewer than 8 bytes long, it is left-justified and padded to the right with blanks. The system also returns 8 into LEN.

Example: To request the management class of the specified SMS-managed data set, code:

```
KEY    #      LEN    PARM
C005   0001   0008   - - - - - - - -
```

Return data class specification - Key = 'C006'

DINRDACL requests the data class of the specified SMS-managed data set. When you code DINRDACL, # must be 1, LEN must be at least 8 and can be longer, and the PARM field must be the length specified by the value in LEN.

If a data class is associated with the data set, the system returns the data class identifier into PARM. If the identifier is fewer than 8 bytes long, it is left-justified and padded to the right with blanks. The system also returns 8 into LEN.

Example: To request the data class of the specified data set, code:

```
KEY    #      LEN    PARM
C006   0001   0008   - - - - - - - -
```

Return record organization specification - Key = 'C00B'

DINRRECO requests the organization of the records in the specified VSAM data set. When you code DINRRECO, # must be 1, LEN must be at least 1 and can be longer, and the PARM field must be the length specified by the value in LEN.

The record organization is available only when the data set is first allocated (that is, when the data set has a disposition of NEW). If the record organization is available, the system returns the record organization into PARM. The system also returns 1 into the LEN field. The value returned in PARM is one of the following:

X'80' VSAM key-sequenced data set (KS)

X'40' VSAM entry-sequenced data set (ES)

X'20' VSAM relative record data set (RR)

X'10' VSAM linear space data set (LS)

Example: To determine how records are organized in the specified VSAM data set, code:

```
KEY    #      LEN    PARM
C00B   0001   0001   -
```

Return key offset specification - Key = 'C00C'

DINRKEYO requests the key offset. The key offset is the position of the first byte of the key in each logical record of the specified VSAM data set. If the key is at the beginning of the logical record, the offset is zero. When you code DINRKEYO, # must be 1, LEN must be at least 4 and can be longer, and the PARM field must be the length specified by the value in LEN.

The system returns into PARM a 4 byte hexadecimal number representing the offset of the key. If there is a key offset associated with the data set, it stores 4 into LEN. The value of the offset is less than or equal to 65535 bytes.

Example: To determine the key offset in a record of the specified VSAM data set, code:

```
KEY    #      LEN    PARM
C00C   0001   0004   - - - -
```

Return copy DD specification - Key = 'C00D'

DINRREFD requests the name of the JCL DD statement from which the attributes of the specified data set were copied. When you code DINRREFD, # must be 1, LEN must be at least 26 and can be longer, and the PARM field must be the length specified by the value in LEN.

If there is a JCL DD statement from which attributes were copied, the system returns the name of the DD statement into PARM. The name can be a *ddname*, a *stepname.ddname*, or a *stepname.procstepname.ddname* where *ddname* is the label on a JCL DD statement, and *stepname* and *procstepname* are labels that appear on JCL EXEC statements. If the name is fewer than 26 bytes long, it is left-justified and padded to the right with blanks. The system also returns 26 into LEN.

Example: To determine the name of the JCL DD statement from which the attributes of the specified data set were copied, code:

```
KEY    #      LEN    PARM
C00D   0001   001A   - - -...
```

Return copy profile specification - Key = 'C00E'

DINRSECM requests the name of the RACF profile from which the RACF profile of the specified data set was copied, and to optionally request an indication of whether the profile was defined generically.

The first LEN and PARM pair in the text unit return the dsname of the profile to be copied. If coded, the second LEN and PARM pair return an indication of whether the profile was defined generically, such as a profile defined with the GENERIC subparameter on the JCL DD statement SECMODEL parameter.

When you code DINRSECM, # is the number of LEN and PARM pairs, LEN1 at least 44 bytes long, and can be longer, and PARM1 is a field the length of the value specified in LEN1.

If coded, LEN2 must be at least one byte, and can be longer, and PARM2 is a field the length of the value specified in LEN2.

If there is a copy profile associated with the data set, the system returns the length of the dsname into LEN1 and the dsname into PARM1. If the name is fewer than 44 bytes long, it is left-justified and padded to the right with blanks. The system also returns 44 into LEN.

If LEN2 and PARM2 are coded, the system also returns an indication of whether the profile is generic. For a generic profile, the system returns 1 in LEN2 and X'80' in PARM2. For a profile that is not generic, the system returns 0 in LEN2.

Example: To determine the dsname of the RACF profile that was used to supply the profile of the specified data set, code:

```
KEY    #      LEN1   PARM1
C00E   0001   002C   - - - ...
```

Example: To determine the dsname of the RACF profile that was used to supply the profile of the specified data set, and to request and indication of whether the data set is generic, code:

```
KEY    #      LEN1   PARM1         LEN2   PARM2
C00E   0002   002C   - - - ...     0001   -
```

Return copy model specification - Key = 'C00F'

DINRLIKE requests the name of the model data set from which the attributes of the specified data set were copied.

When you code DINRLIKE, # must be 1, LEN must be at least 44 and can be longer, and the PARM field must be the length specified by the value in LEN. If there is a model data set associated with the data set, the system returns the dsname into PARM. If the dsname is fewer than 44 bytes long, it is left-justified and padded to the right with blanks. The system also returns 44 into LEN.

Example: To determine the dsname of the model data set from which the attributes of the specified data set were copied, code:

```
KEY    #      LEN    PARM
C00F   0001   002C   - - - ...
```

Return average record specification - Key = 'C010'

DINRAVGR requests the allocation unit that was used when the specified data set was allocated. When you code DINRAVGR, # must be 1, LEN must be at least 1 and can be longer, and the PARM field must be the length specified by the value in LEN.

If there is an average record length associated with the data set, the system returns a code into PARM. The system also returns 1 into LEN. The code returned in PARM is one of the following:

X'80' Single-record units (U)

X'40' Thousand-record units (K)

X'20' Million-record units (M)

Example: To determine the unit of allocation used to allocate the specified data set, code:

```
KEY    #      LEN    PARM
C010   0001   0001   - - - ...
```

Return data set type specification - Key = 'C012'

Code DINRDSNT to request the return of the data set type.

When you code DINRDSNT, # must be 1 and LEN must be at least 1 and can be longer; the PARM field must be the length specified by the value in LEN. If a data set type specification is associated with the allocation, the system returns the data set type in PARM and 0001 in LEN. PARM will contain one of the following:

X'80' Represents a library (PDSE).

X'40' Represents a partitioned data set (PDS).

X'20' Represents a FIFO special file (PIPE).

X'10' Represents an HFS data set (HFS). This type of UNIX file system is different from a z/OS File System (zFS) that resides in a linear data set.

X'08' Represents an extended format required data set (EXTREQ).

X'04' Represents an extended format preferred data set (EXTPREF).

X'02' Represents a basic format data set (BASIC).

X'01' Represents a large format data set (LARGE = greater than 65,535 tracks).

Example: To request the data set type, code:

```
KEY    #      LEN    PARM
C012   0001   0001   - - - ...
```

Note: For this text unit, the information specified by the user in JCL or via dynamic allocation text units is retrieved via the Scheduler JCL Facility (SJF). If the information was not available, no information will be returned.

Return spin data set specification - Key = 'C013'

Code DINRSPIN to request the return of the spin data set specification.

When you code DINRSPIN, # must be 1, LEN must be at least 1, and can be longer, and the PARM field must be the length specified by the value in LEN.

If there is a spin data set specification associated with the data set, the system returns 1 into LEN and one of the following into PARM:

X'80' Data set available for printing when it is unallocated

X'40' Data set available for printing at the end of the job

Example: To request the spin data set specification, code:

```
KEY    #     LEN   PARM
C013   0001  0001  -
```

Return segment spin data set specification - Key = 'C014'

Code DINRSEGM to request the return of the number of logical, line-mode pages (the segment) to be produced for a SYSOUT data set before the segment becomes eligible for immediate printing. When you code DINRSEGM, # must be 1, LEN must be at least 4, and the PARM field must be the length specified by the value in LEN.

If there is a segment spin specification associated with the data set, the system returns a positive hexadecimal integer from X'00000001' to X'0001869F' (equivalent decimal value from 1 to 99,999) into PARM. The system also returns 4 into LEN.

Example: To request the return of the value of the segment, code:

```
KEY    #     LEN   PARM
C014   0001  0004  - - - -
```

Note that for text unit keys 'C003' = 'C014' PARM remains unchanged if no data is available for the request.

Return z/OS UNIX file path specification - Key = 'C017'

DINRPATH requests the return of the path specification for the z/OS UNIX file associated with the specified allocation.

When you code DINRPATH, # must be 1 and LEN must be at least the length of the pathname and can be longer, and the PARM field must be the length specified by the value in LEN.

If a pathname is associated with the allocation, the system returns the pathname in PARM and its length in LEN.

Example: To request the return of the z/OS UNIX file pathname for ddname MYDD, code:

Input:

```
KEY    #     LEN   PARM
0001   0001  0004  D3 E8 C4 C4
C017   0001  00FF  00  00...00...00...00...
```

Output:

```
KEY    #     LEN   PARM
C017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
```

The retrieved output indicates that pathname `/u/myuid/myapp/scr.dat` is associated with MYDD.

Return z/OS UNIX file options - Key = 'C018'

DINRPOPT requests the return of the options for the z/OS UNIX file associated with the specified allocation.

When you code DINRPOPT, # must be 1 and LEN must be at least 4 and can be longer, and the PARM field must be the length specified by the value in LEN.

If there are file options associated with the allocation, the system returns the file options in PARM and 0004 in LEN. PARM will contain one of the following values or an “inclusive OR” result from any combination of the following values:

Table 94. DINRPOPT Options

Four-byte Hex Integers	JCL Equivalent
X'00000080'	OCREAT
X'00000040'	OEXCL
X'00000020'	ONOCTTY
X'00000010'	OTRUNC
X'00000008'	OAPPEND
X'00000004'	ONONBLOCK
X'00000003'	ORDWR
X'00000002'	ORDONLY
X'00000001'	OWRONLY

Example: To request the return of the z/OS UNIX file options for **/u/myuid/myapp/scr.dat**, code:

Input:

```
KEY    #     LEN   PARM
8017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
C018   0001  0004  00000000
```

Output:

```
KEY    #     LEN   PARM
C018   0001  0004  000000C3
```

The retrieved information indicates a file access mode that is "create the file if it does not already exist with the intent to read and write the file" (the "inclusive OR" expression for OCREAT+OEXCL+ORDWR).

Return z/OS UNIX file access attributes - Key = 'C019'

DINRPMDE requests the return of the file options for the z/OS UNIX file being dynamically allocated.

When you code DINRPMDE, # must be 1 and LEN must be at least 4 and can be longer, and the PARM field must be the length specified by the value in LEN.

If there are file options associated with the allocation, the system returns the file options in PARM and 0004 in LEN. PARM will contain one of the following values or an "inclusive or" result from any combination of the following values:

Table 95. DINRPMDE Attributes

Four-byte Hex Integers	JCL Equivalent
X'00000800'	SISUID
X'00000400'	SISGID
X'00000100'	SIRUSR
X'00000080'	SIWUSR
X'00000040'	SIXUSR
X'000001C0'	SIRWXU
X'00000020'	SIRGRP
X'00000010'	SIWGRP
X'00000008'	SIXGRP
X'00000038'	SIRWXG
X'00000004'	SIROTH
X'00000002'	SIWOTH
X'00000001'	SIXOTH
X'00000007'	SIRWXO

Example: To request the return of the z/OS UNIX file access attributes for **/u/myuid/myapp/scr.dat**, code:

Input:

```
KEY    #     LEN   PARM
8017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
C019   0001  0004  00000000
```

Output:

```
KEY    #     LEN   PARM
C019   0001  0004  000000A0
```

The retrieved information indicates a file access attribute allowing the owner of the file to write the file and users in the same file group class to read the file (the "inclusive OR" expression for SIWUSR+SIRGRP).

Return z/OS UNIX file normal disposition specification - Key = 'C01A'

DINRPNDS requests the return of the normal disposition for the z/OS UNIX file.

When you code DINRPNDS, # must be 1 and LEN must be at least one and can be longer, and the PARM field must be the length specified by the value in LEN.

If there is a normal file disposition associated with the allocation, the system returns the disposition in PARM and 0001 in LEN. PARM will contain one of the following values:

- X'04' - the file is to be deleted
- X'08' - the file is to be kept

Example: To request the return of the z/OS UNIX file normal disposition for **/u/myuid/myapp/scr.dat**, code:

Input:

```
KEY    #     LEN   PARM
8017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
C01A   0001  0001  00
```

Output:

```
KEY    #     LEN   PARM
C01A   0001  0001  04
```

The retrieved information indicates a normal disposition of delete for the file.

Return z/OS UNIX file abnormal disposition specification - Key = 'C01B'

DINRPCDS requests the return of the abnormal disposition for the z/OS UNIX file.

When you code DINRPCDS, # must be 1 and LEN must be at least one and can be longer, and the PARM field must be the length specified by the value in LEN.

If there is an abnormal file disposition associated with the allocation, the system returns the disposition in PARM and 0001 in LEN. PARM will contain the following values:

- X'04' - the file is to be deleted
- X'08' - the file is to be kept

Example: To request the return of the z/OS UNIX file abnormal disposition for **/u/myuid/myapp/scr.dat**, code:

Input:

```
KEY    #     LEN   PARM
8017   0001  0016  61 A4 61 94 A8 A4 89 84 61 94 A8 81 97 97 61 A2 83 99 4B 84 81 A3
C01B   0001  0001  00
```

Output:

```
KEY    #     LEN   PARM
C01B   0001  0001  08
```

Note: The conditional disposition is KEEP for this file.

Return organization of a z/OS UNIX file - Key = 'C01D'

DINRFDAT requests the return of the organization of an z/OS UNIX file.

When you code DINRFDAT, # must be 1, LEN must be at least 1 and can be longer, and the PARM field must be the length specified by the LEN value. If there is an organization associated with the allocation, the system returns the organization in PARM and 0001 in LEN. PARM will contain one of the following values:

- X'80' - the z/OS UNIX file organization is binary — records not delimited.
- X'40' - the z/OS UNIX file is text — records delimited by EBCDIC newline characters (x'15').
- X'20' - the z/OS UNIX file is organized into records that have prefixes. Each record prefix contains the length of the record that follows. On output, the access method inserts a record prefix at the beginning of each record. On input, the access method uses the record prefix to determine the length of each record. The access method does not return the prefix as part of the record. Code FILEDATA=RECORD when you cannot code FILEDATA=TEXT because your data might contain bytes that are considered delimiters.

Example: To request the return of the organization of the z/OS UNIX file **/u/myuid/scr.dat**, code:

Input:

```
KEY    #     LEN   PARM
8017   0001  0010  61 A4 61 94 A8 A4 89 84 61 A2 83 99 4B 84 81 A3
C01D   0001  0001  00
```

Output:

```
KEY    #     LEN   PARM
C01D   0001  0001  80
```

The retrieved information indicates a file organization of binary.

Return spin of interval specification - Key = 'C02A'

Code DINRSPI2 to request the return of the spin interval specification.

When you code DINRSPI2, # must be 1, LEN must be 3 to 8 and the PARM field is the character (EBCDIC) representation of the SPIN interval, such as F9F9F9 for '999' or F1F27AF0F0 for '12:00'. If a spin data set specification is associated with the data set, the system returns the PARM length into LEN and the string specified for DALSPI2 (or the second parameter of the SPIN keyword value on JCL DD statement) in PARM: See “SPIN interval for the allocated SYSOUT data set - Key = '802A'” on page 662 for valid strings.

Example: To request the return of the SPIN interval for the DD statement with name **AA1**, code:

Input:

```
KEY    #       LEN   PARM
0001   0001    0008  C1 C1 F1 40 40 40 40 40
C02A   0001    0008
```

Output:

```
KEY    #       LEN   PARM
0001   0001    0008  C1 C1 F1 40 40 40 40 40
C02A   0001    0005  F0 F1 7A F0 F0
```

Note that the string is not validated until passed to the job entry subsystem responsible for the SYSOUT allocation.

Return symbol list on a DD statement - Key = 'C02B'

Code DINSYML to request the list of symbol names to be passed to JES2 when an internal reader is allocated.

When you code DINSYML, # must be the number of symbol names to be returned, from 1 to a maximum of 128 (X'0080'). If # is less than the number of symbol names that were requested for the DD, only the number of symbol names requested with # are returned, with no indication that there were additional symbol names that were not returned. For each value to be returned, LEN should be 8, and upon return the PARM field will contain the symbol name. Be aware that if a symbol name is shorter than 8 characters, the returned length may be 8 and the PARM field may be padded with blanks.

Example: To request the return of two symbol names for the DD statement with name DD1, code:

Input:

```
KEY    #       LEN   PARM
0001   0001    0003  C4 C4 C1
C02B   0002    0008
               0008
```

Output:

```
KEY    #       LEN   PARM
0001   0001    0003  C4 C4 C1
C02B   0002    0008  D5 C1 D4 C5 E2 40 40 40
               0008  C1 C2 C3 40 40 40 40 40
```

Return DSNTYPE version information - Key = 'C02C'

Code DINRDSNV to retrieve the data set type version information.

When you code DINRDSNV, # must be 1, LEN must be 1, and the PARM field will contain the DSNTYPE version number.

Example: To request the return of the data set type and version for the DD statement with name DD1, code:

Input:

```
KEY    #       LEN   PARM
0001   0001    0003  C4 C4 C1
C012   0001    0001
C02C   0001    0001
```

Output:

```
KEY    #       LEN   PARM
0001   0001    0003  C4 C4 C1
C012   0001    0001  80
C02C   0001    0001  02
```

In this example, the data set type represents a library (PDSE) version 2.

Note: For this text unit, the information specified by the user in JCL or via dynamic allocation text units is retrieved via the Scheduler JCL Facility (SJF). If the information was not available, no information is returned.

Return MAXGENS generation information - Key = 'C02D'

When you have applied APAR OA42358, code DINRMAXG to retrieve the number of generations of members to keep.

When you code DINRMAXG, # must be 1, LEN must be 1, length must be 4, and the PARM field will contain the number of generations to be kept.

Example: To request the number of generations to be kept for the DD statement with name DD1, code:

Input:

```
KEY    #       LEN   PARM
0001   0001    0003  C4 C4 C1
C02D   0001    0001
```

Output:

```
KEY    #       LEN   PARM
0001   0001    0003  C4 C4 C1
C02D   0001    0001  05
```

In this example, the data set type represents a library (PDSE) version 2.

Note: For this text unit, the information specified by the user in JCL or via dynamic allocation text units is retrieved via the Scheduler JCL Facility (SJF). If the information was not available, no information is returned.

For more information, refer to *z/OS Summary of Message and Interface Changes*.

Return GDGORDER specification – Key = 'C02E'

Code DINRGDGO to request the return of the order in which the individual generation data sets (GDSs) are concatenated.

When you code DINRGDGO, # must be 1, LEN must be 1, and upon return, the PARM value will contain one of the following values:

X'80' USECATLG – The GDS concatenation is ordered as specified in the GDG data set catalog entry.

X'40' LIFO – The GDS concatenation is ordered with the newest GDS defined first and the oldest GDS last.

X'20' FIFO – The GDS concatenation is ordered with the oldest GDS defined first and the newest GDS last.

Example: To request the return of the specification of GDGORDER on a batch or dynamically allocated DD which specified GDGORDER=FIFO on the request, code:

Input:

```
KEY    #       LEN   PARM
C02E   0001    0001  00
```

Output:

```
KEY    #       LEN   PARM
C02E   0001    0001  20
```

Note: For this text unit, the information specified by the user in JCL or via dynamic allocation text units is retrieved via the Scheduler JCL Facility (SJF). If the information was not available, no information is returned. Also, this key does not return the value of the GDG order setting in the catalog entry for the GDG.

Example of a Dynamic Allocation Request

The example in Figure 69 on page 717 is a dynamic allocation request allocating SYS1.LINKLIB with a status of SHARE. It also requests that dynamic allocation returns the ddname associated with SYS1.LINKLIB.

Figure 70 on page 718 shows the parameter list that is built from the DYNALLOC invocation in Figure 69 on page 717.

```
DYN CSECT
 USING *,15
 STM 14,12,12(13)
 BALR 12,0
BEGIN DS 0H
 USING BEGIN,12
 LA 0,50                 AMOUNT OF STORAGE REQUIRED FOR THIS REQUEST.
 GETMAIN R,LV=(0)        GET THE STORAGE NECESSARY FOR THE REQUEST.
 LR 8,1                  SAVE THE ADDRESS OF THE RETURNED STORAGE.
 USING S99RBP,8          ESTABLISH ADDRESSABILITY FOR S99RBP DSECT.
 LA 4,S99RBPTR+4         POINT FOUR BYTES BEYOND START OF S99RBPTR.
 USING S99RB,4           ESTABLISH ADDRESSABILITY FOR RB DSECT.
 ST 4,S99RBPTR           MAKE 'RBPTR' POINT TO RB.
 OI S99RBPTR,S99RBPND    TURN ON THE HIGH-ORDER BIT IN RBPTR.
 XC S99RB(RBLEN),S99RB   ZERO OUT 'RB' ENTIRELY.
 MVI S99RBLN,RBLEN       PUT THE LENGTH OF 'RB' IN ITS LENGTH FIELD.
 MVI S99VERB,S99VRBAL    SET THE VERB CODE FIELD TO ALLOCATION FUNCTION.
 LA 5,S99RB+RBLEN        POINT PAST 'RB' TO START OF TUP LIST.
 USING S99TUPL,5         ESTABLISH ADDRESSABILITY FOR TEXT UNIT PTRS.
 ST 5,S99TXTPP           STORE ADDRESS OF TUP LIST IN THE RB.
 LA 6,DSNTU              GET ADDRESS OF FIRST TEXT UNIT
 ST 6,S99TUPTR           AND STORE IN TUP LIST.
 LA 5,S99TUPL+4          GET ADDRESS OF NEXT TUP LIST ENTRY.
 LA 6,STATUSTU           GET ADDRESS OF SECOND TEXT UNIT
 ST 6,S99TUPTR           AND STORE IN TUP LIST.
 LA 6,S99TUPL+8          POINT PAST END OF TUP LIST.
 USING S99TUNIT,6        ESTABLISH ADDRESSABILITY TO TEXT UNIT.
 LA 5,S99TUPL+4          GET ADDRESS OF NEXT TUP LIST ENTRY.
 ST 6,S99TUPTR           STORE ADDRESS OF TEXT UNIT IN TUP LIST.
 OI S99TUPTR,S99TUPLN    TURN ON HIGH-ORDER BIT IN LAST TUP LIST ENTRY.
 MVC S99TUNIT(14),RETDDN   MOVE RETURN DDNAME TEXT UNIT TO PARM AREA.
 LR 1,8                  PUT ADDRESS OF REQUEST BLOCK POINTER IN REG 1.
         DYNALLOC        INVOKE DYNALLOC TO PROCESS THE REQUEST.
 LM  14,12,12(13)
         BR  14          RETURN TO CALLER.
RBLEN    EQU (S99RBEND-S99RB)
DSNTU    DC AL2(DALDSNAM)
         DC X'0001'
         DC X'000C'
         DC C'SYS1.LINKLIB'
STATUSTU DC AL2(DALSTATS)
         DC X'0001'
         DC X'0001'
         DC X'08'
RETDDN   DC AL2(DALRTDDN)
         DC X'0001'
         DC X'0008'
         DS CL8
         IEFZB4D0
         IEFZB4D2
DYN      CSECT
         END
```

Figure 69. Example of a Dynamic Allocation Request

Note the concepts that the example illustrates:

- You need to request storage via the GETMAIN or STORAGE macro for the request block and the DALRTDDN text unit, because DYNALLOC modifies them. The DALDSNAM and DSLSTATS text units can be in static storage in your program.

  In the example, the GETMAIN request is for 50 bytes, derived as follows:

Bytes	Purpose
4	Pointer to the request block.

20 Request block space.

12 Four bytes each for three text unit pointers.

14 Text unit space for the requested return of the ddname.

- IEFZB4D0 provides DSECTs that map the parameter list structure.
- The example uses IEFZB4D2 mnemonics in the text unit keys. For a complete list of the fields mapped by the IEFZB4D2 mapping macro, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Figure 70 shows the parameter list that results from the code in Figure 69 on page 717. It is the request block structure needed to allocate data set SYS1.LINKLIB with a disposition of SHARE, and to return the ddname assigned by DYNALLOC.

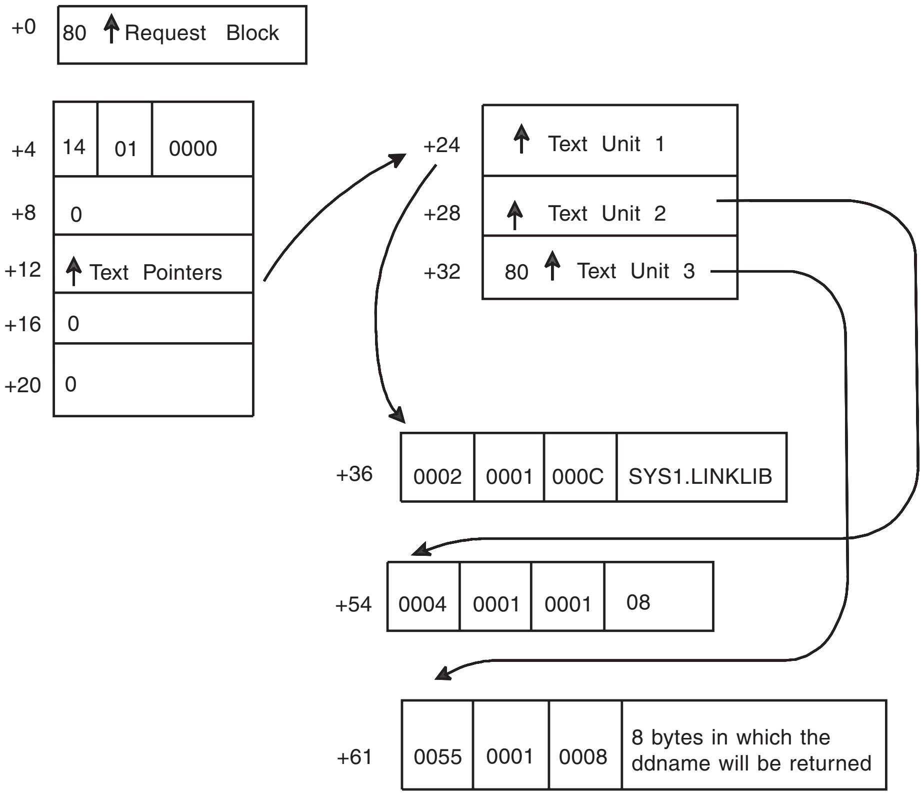


Figure 70. Parameter List Resulting from Dynamic Allocation Example

Chapter 27. Dynamic output

Note: You can use dynamic output in a JES2 environment or a JES3 4.2.1 or later environment.

Before a program writes to a system output (SYSOUT) data set, the program must:

1. Describe the processing options for the SYSOUT data set. This means giving the system information such as what kind of print paper to use, how many copies, how many lines on a page, and so forth.
2. Allocate the data set.
3. Open the data set.

The first step in the sequence, describing the SYSOUT processing options for the data set, can be accomplished before the program executes by using an OUTPUT JCL statement. However, by using dynamic output, a program can describe its SYSOUT processing options during program execution.

The second step in the sequence, allocating the data set, can also be accomplished in two ways. Before execution, you can allocate by using the DD JCL statement. During execution, you can allocate by using dynamic allocation.

To accomplish both steps dynamically, use dynamic allocation with dynamic output. When you use them together, the available processing options are similar to the options available through the OUTPUT and DD JCL statements. For example, you can use dynamic output with dynamic allocation to describe the same SYSOUT characteristics that you could describe by using the following JCL:

```
//OUT1    OUTPUT COPIES=10,DEST=FRANCE
//DD1     DD     SYSOUT=A,OUTPUT=*.OUT1
```

Dynamic output has two advantages:

- ***Flexibility*** — Dynamic output lets the application change the SYSOUT processing options based on input data to the application when the application executes. It provides more flexibility than the OUTPUT JCL statement, which requires you to specify the SYSOUT processing options before the application executes.
- ***Convenience*** — Dynamic output allows you to write your application program so that it changes your SYSOUT processing options based on the input your program receives. This eliminates the need to change your JCL OUTPUT statement for each of your SYSOUT processing requirements.

Dynamic output affects only SYSOUT data sets. The programs that invoke dynamic output can be authorized or unauthorized. To use dynamic output, issue the OUTADD and OUTDEL macros.

For information on the OUTPUT JCL statement, see *z/OS MVS JCL Reference*. For general information on JCL, see *z/OS MVS JCL User's Guide*.

Note: BPXWDYN is a text interface to a subset of the SVC 99 (dynamic allocation) and SVC 109 (dynamic output) services. BPXWDYN supports data set allocation, unallocation, concatenation, and the addition and deletion of output descriptors. BPXWDYN is designed to be called from REXX, but it may be called from several other programming languages, including Assembler, C, and PL/I. For more information, see *z/OS Using REXX and z/OS UNIX System Services*.

Creating and naming output descriptors

When you issue the OUTADD macro, the system creates an ***output descriptor***. An output descriptor represents your SYSOUT processing options. (See “Output descriptors and text units” for an explanation of output descriptors.) When you issue OUTADD, you either specify a name for the descriptor, or allow the system to assign one. The name must be unique in the present job step.

The name allows you to reference one or more output descriptors when you invoke dynamic allocation. (See “OUTPUT statement reference - Key = '8002'” on page 651.) These descriptors can be created either by the OUTADD macro or by the OUTPUT JCL statement, as long as they exist when dynamic allocation is invoked. If you allocate a SYSOUT data set with a DD JCL statement, you must define the descriptor with an OUTPUT JCL statement. If you allocate a SYSOUT data set dynamically, you can define the descriptor with the OUTPUT JCL statement or use dynamic output.

System generated names

In a program that creates a great number of output descriptors, it might be more convenient to use system-generated names. A system-generated descriptor name has the form SYSxxxxx, where xxxxx is any character string in the range 00001 through 32767. If you delete an output descriptor with a system-generated name, the system can generate the same name again after wrapping completely around another 32767 name generations. However, when the name comes up again, it does not refer to the deleted descriptor, which remains deleted.

Job step considerations

Once an output descriptor name exists in a job step, the job step cannot have another descriptor with the same name. Thus, if a descriptor name is produced within a job step by an OUTPUT JCL statement or by an invocation of dynamic output, you cannot invoke OUTADD in that job step to create another descriptor with the same name. An output descriptor that is created dynamically (by OUTADD) in a particular job step cannot be referenced outside the job step.

When you use OUTADD to create output descriptors in a program that also uses checkpoint/restart, you must observe the restrictions that are described in *z/OS DFSMSdfp Checkpoint/Restart*.

Output descriptors and text units

An output descriptor, which the system creates or deletes when you invoke dynamic output, includes one or more ***text units***. Each text unit represents a set of SYSOUT processing options. Thus, the output descriptor represents all the SYSOUT processing options associated with an invocation of dynamic output.

To select particular SYSOUT processing options, you must code the corresponding text units. A text unit consists of at least four consecutive fields:

key: A two-byte field that identifies the SYSOUT processing options.

count: A two-byte field that specifies the number of length/value field pairs.

length:
A two-byte field that specifies the size of the value field.

value: A variable length data field that qualifies the SYSOUT processing option.

Figure 71 shows a group of text units and the text unit pointers that reference them. The leftmost bit of the last pointer must be 1.

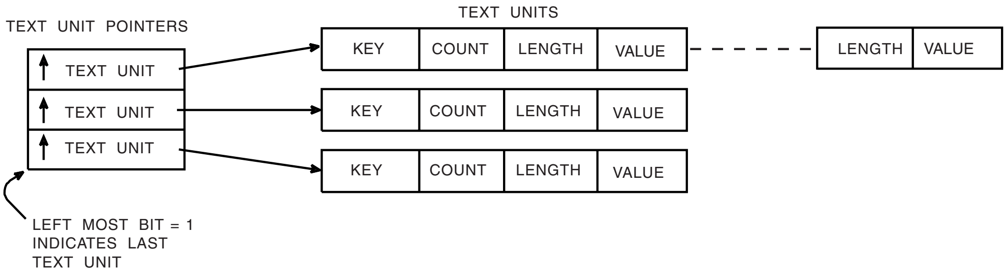


Figure 71. Text Units and Text Unit Pointers

For an invocation of dynamic output, the text unit keys must be unique. To be unique, a text unit cannot have the same key that another text unit has. If two or more text units have the same key, dynamic output returns a return code X'0C' and a reason code X'30E'.

A text unit can have one or more length/value field pairs. Each length/value field pair represents a SYSOUT processing option. The following example shows a text unit that has one value field:

```
DC XL2'0001'       KEY FOR BURST OPTION
DC XL2'0001'       COUNT OF VALUE FIELDS
DC XL2'0001'       LENGTH OF VALUE FIELD
DC XL1'02'         VALUE FIELD: BURST=YES
```

For some keys, the length/value field may be repeated. The following example shows a text unit that has two length/value fields:

```
DC XL2'0002'       KEY FOR 'CHARS' OPTION
DC XL2'0002'       COUNT OF VALUE FIELDS=2
DC XL2'0004'       LENGTH OF FIRST VALUE FIELD
DC CL4'TAB1'       FIRST VALUE FIELD
DC XL2'0004'       LENGTH OF SECOND VALUE FIELD
DC CL4'TAB9'       SECOND VALUE FIELD
```

If you need a DSECT to define text unit fields symbolically, use the IEFDOTUM mapping macro described in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Table of dynamic output text units and JCL equivalents

The OUTADD macro is a dynamic version of the OUTPUT JCL statement. You can use Table 96 on page 724 as a quick reference when you are coding text units. This figure defines the OUTPUT JCL keyword parameters and their equivalent dynamic output text units. For example, if you wanted the ADDRESS function, you would do the following:

1. Locate ADDRESS in the first table column
2. Find the corresponding key that appears in the third table column - X'0027'
3. Code X'0027' as the first field of the text unit
4. Code the length and value fields as defined in columns 4 - 6

When a JCL keyword parameter corresponds to more than one text unit, there is a row in the table for each corresponding text unit. The subparameters corresponding to the text unit appear in parentheses following the keyword name in column one.

Relationship between text units and JCL equivalents

While "Table of dynamic output text units and JCL equivalents" on page 721 describes the correspondence between dynamic output text units and OUTPUT JCL in general terms, this information describes that correspondence in more detail.

If you need more information about a particular processing option, see the information on the OUTPUT statement in the *z/OS MVS JCL Reference*.

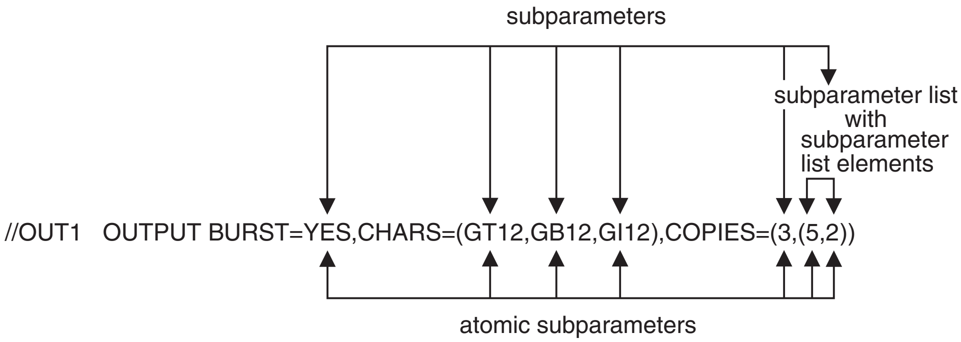

subparameters
subparameter list with subparameter list elements
//OUT1 OUTPUT BURST=YES,CHARS=(GT12,GB12,GI12),COPIES=(3,(5,2))
atomic subparameters

Figure 72. Sample OUTPUT JCL Statement

OUTPUT JCL statement: Figure 72 shows a sample Output JCL statement with various keywords and subparameters. These elements are used as examples in the following discussion.

The OUTPUT JCL statement contains keyword parameters (for example, BURST=), which are referred to in this information simply as keywords. To the right of the keyword's equal sign is variable information consisting of subparameters. This variable information can have one of the following forms:

- **One subparameter** (for example YES)
- **More than one subparameter** (for example GT12,GB12,GI12)

  In this case, the subparameters are separated by commas and enclosed in parentheses.

Subparameters themselves can consist of one or more parameters. The following subparameter forms are possible:

- **One parameter** (for example 3)
- **More than one parameter** (for example (5,2))

  A subparameter that can consist of more than one parameter is called a subparameter list. The parameters contained in a subparameter list are called subparameter list elements. These subparameter list elements are separated by commas and enclosed in parentheses.

A subparameter list element and a subparameter that is not a subparameter list are referred to as **atomic subparameters**.

Text units: You can specify the same SYSOUT processing options that are available through an OUTPUT JCL keyword by using one or more dynamic output text units. Each text unit provides the same SYSOUT processing options as one or more subparameters of a particular OUTPUT JCL keyword.

To see how a particular OUTPUT JCL keyword corresponds to a dynamic output text unit, refer to the following columns in Table 96 on page 724.

Table Column 1 — JCL Keyword Parameter
Use this column to locate the keyword that you would like to specify as a dynamic output text unit. For most keywords, you can use a single text unit to specify all the SYSOUT processing options available through the keyword. For these keywords, the keyword and its corresponding text unit are shown in a single row in the table. Some keywords, however, have more than one corresponding text unit. For these keywords, the atomic subparameters corresponding to a single text unit are shown in a single row of the table. For example, the COPIES *data set count* subparameter is specified by text unit key X'0009' and is shown in a single table row. The COPIES *group values* subparameter is specified by text unit key X'000A' and is shown in another separate table row.

Table Column 2 — Symbolic Key from IEFDOKEY Macro
Use this column to find the symbolic key that identifies the text unit.

Table Column 3 — Key in Hex
Use this column to find the text unit's hexadecimal key. This key serves as the text unit's unique identifier, and is equivalent to the value of the symbolic key in column 2.

Columns 4, 5 and 6 define the length/value pairs for each text unit. Within a text unit, each length/value pair corresponds to an atomic subparameter of the text unit's corresponding OUTPUT JCL keyword. The length/value pair specifies the same SYSOUT processing option as its corresponding atomic subparameter. Figure 73 shows an example of a keyword with two corresponding text units. In this example the COPIES keyword corresponds to the data set count and group values text units. The data set count text unit has one length/value pair, while the group values text unit has two length value pairs. Note that there is one length/value pair for each atomic subparameter.

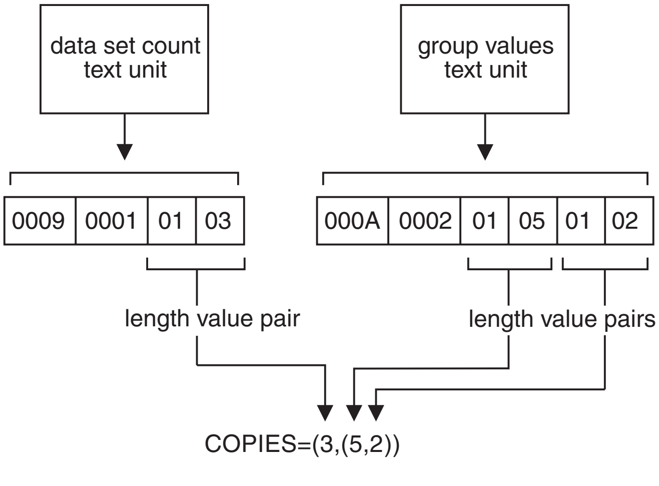


Figure 73. COPIES Keyword and its Dynamic Output Text Units

Table column 4 — Maximum number of value fields
Use this column to determine the maximum number of length/value pairs that you can code for a text unit. When a text unit can have more than one

length/value pair, the length/value pairs must be in the same order as their corresponding atomic subparameters. For example, in Figure 72 on page 722, the first through fourth subparameters of the CHARS keyword correspond, respectively, to the first through fourth length/value pairs of the text unit with key X'0002'.

Table column 5 — Length of Value Field

Use this column to determine the length of the value field in a text unit's length/value pair. If the value field contains character data, you can use any length within the minimum and maximum lengths defined. If, however, the value field does not contain character data, the length field must be equal to the length of the value field.

Alternatively, you can have a zero length pair to act as a place holder. A zero length is allowed whenever the length value pair's corresponding atomic subparameter may be specified as null. Using a length of zero indicates a null value for the length/value pair and is equivalent to having a null atomic subparameter in JCL. For example, in ADDRESS=(ZOBRE,,POK) the second subparameter is null.

Table column 6 — Value field

Use this column to determine what types of values (for example, EBCDIC text or binary numbers) are allowed for the value field of a length/value pair. The value field of a length/value pair has the same syntax as its corresponding atomic subparameter. However, the value field does not use JCL delimiters, such as commas and delimiting apostrophes, which are not part of the actual atomic subparameter. Also, for most atomic subparameters whose values are a choice of character string constants (for example, YES or NO), the corresponding text unit value field uses a 1-byte hexadecimal number instead of a character string constant. For example, in Figure 72 on page 722, BURST=YES is specified by a text unit with key X'0001' and a value field X'02'. This column defines value fields that use hexadecimal numbers in place of character string constants and describes any other deviations that might exist between a value field's syntax and the syntax of its corresponding atomic subparameter.

Table 96. Dynamic Output Text Units and their JCL Equivalents

JCL Keyword Parameter	*Symbolic key from IEFDOKEY Macro*	*Key in Hex*	*Maximum number of value fields*	*Length of value field*	*Value field*	*Function*
ADDRESS	DOADDRES	0027	4	0-60	EBCDIC text characters	Specifies the delivery address for the SYSOUT data set.
AFPPARMS	DOAFPPRM	0051	1	1-54	Cataloged data set name	Identifies a data set containing control parameters for the AFP Print Distributor feature of PSF.
AFPSTATS	DOAFPST	0048	1	1	X‘40’ for YES X‘80’ for NO	Specifies to Print Services Facility (PSF) that an AFP Statistics report is to be generated while printing this SYSOUT data set.
BUILDING	DOBUILD	0028	1	1-60	EBCDIC text characters	Specifies the building location associated with the SYSOUT data set.
BURST	DOBURST	0001	1	1	X‘02’ for YES X‘04’ for NO	Directs output to a stacker on a 3800 Printing Subsystem.

Table 96. Dynamic Output Text Units and their JCL Equivalents (continued)

JCL Keyword Parameter	*Symbolic key from IEFDOKEY Macro*	*Key in Hex*	*Maximum number of value fields*	*Length of value field*	*Value field*	*Function*
CHARS	DOCHARS	0002	4	1-4 (see note 1)	Alphanumeric or national (@, $, #) characters	Names character-arrangement tables for printing on a 3800 Printing Subsystem.
CKPTLINE	DOCKPTLI	0003	1	2	binary number from 0 to 32767 decimal	Specifies the maximum lines in a logical page.
CKPTPAGE	DOCKPTPA	0004	1	2	binary number from 1 to 32767 decimal	Specifies the number of logical pages to be printed or transmitted before JES takes a checkpoint.
CKPTSEC	DOCKPTSE	0005	1	2	binary number from 1 to 32767 decimal	Specifies how many seconds of printing are to elapse between each checkpoint of this SYSOUT data set.
CLASS	DOCLASS	0006	1	1	alphanumeric character or *	Assigns the system data set to an output class.
COLORMAP	DOCOLORM	003A	1	1-8	alphanumeric or national (@,$,#) characters	Specifies the AFP resource (object) for the data set that contains color translation information.
COMPACT	DOCOMPAC	0007	1	1-8	alphanumeric characters	Specifies a compaction table for sending this SYSOUT data set to a SNA remote terminal.
COMSETUP	DOCOMSET	0032	1	1-8	alphanumeric characters, $, #, @	Specifies the name of a microfile setup resource.
CONTROL	DOCONTRO	0008	1	1	X'80' for SINGLE X'40' for DOUBLE X'20' for TRIPLE X'10' for PROGRAM	Specifies that all the data records begin with carriage control characters or specifies line spacing.
COPIES (dataset count)	DOCOPIE9	0009	1	1	• For JES2: binary number from 1 to 255 decimal • For JES3: binary number from 0 to 255 decimal	Specifies number of copies printed.
COPIES (group values)	DOCOPIEA	000A	8	1	• For JES2: binary number from 1 to 255 decimal • For JES3: binary number from 1 to 254 decimal	Specifies number of copies printed before next page.
COPYCNT	DOCOPYCN	0052	1	4	0 to 2147483647	Specifies number of copies printed.
DATACK	DODATACK	2022	1	1	X'00' for BLOCK X'80' for UNBLOCK X'81' for BLKCHAR X'82' for BLKPOS	Specifies how errors in printers accessed through the functional subsystem Print Services Facility™ (PSF) are to be reported.
DEFAULT	DODEFAUL	000B	1	1	X'40' for YES X'80' for NO	Specifies that this is a default output descriptor.
DEPT	DODEPT	0029	1	1-60	EBCDIC text characters	Specifies the department identification associated with the SYSOUT data set.
DEST	DODEST	000C	1	1-127	See *z/OS MVS JCL Reference*.	Sends a SYSOUT data set to the specified destination.

Table 96. Dynamic Output Text Units and their JCL Equivalents (continued)

JCL Keyword Parameter	*Symbolic key from IEFDOKEY Macro*	*Key in Hex*	*Maximum number of value fields*	*Length of value field*	*Value field*	*Function*
DPAGELBL	DODPAGEL	0023	1	1	X'40' for YES X'80' for NO	Indicates whether the system should place a security label on each output page. YES means the system should place a label on each page. NO means the system should not place a label on each page.
DUPLEX	DODUPLEX	003D	1	1	X'80' for NO X'40' for NORMAL X'20' for TUMBLE	Specifies whether the job is to be printed on one or both sides of the paper. Overrides comparable FORMDEF specification.
FCB	DOFCB	000D	1	1-4	alphanumeric or national (@, $, #) characters	Specifies FCB image, carriage control tape for 1403 Printer, or data-protection image for 3525 Card Punch.
FLASH (overlay name)	DOFLASE	000E	1	1-4	alphanumeric or national (@, $, #) characters	For printing on a 3800 Printing Subsystem, indicates that the data set is to be printed with forms overlay.
FLASH (count)	DOFLASF	000F	1	1	binary number from 0 to 255 decimal	For printing on a 3800 Printing Subsystem, specifies how many copies are to be printed with forms overlay.
FORMDEF	DOFORMD	001D	1	1-6	alphanumeric or national (@, $, #) characters	Names a library member that PSF uses in printing the SYSOUT data set on a 3800 Printing Subsystem Model 3.
FORMLEN	DOFORMLN	003B	1	1-10	See *z/OS MVS JCL Reference*.	Specifies the form length to be used for a print data set when it is not specified in the DORMDEF parameter.
FORMS	DOFORMS	0010	1	1-8	alphanumeric or national (@, $, #) characters	Identifies forms on which the SYSOUT data set is to be printed or punched.
FSSDATA	DOFSSDAT	0047	1	1-127	EBCDIC text characters	Data that JES ignores but passes to a functional subsystem application.
GROUPID	DOGROUPI	0011	1	1-8	alphanumeric characters	Specifies that this SYSOUT data set belongs to a user-named output group. (JES2 only)
INDEX	DOINDEX	0012	1	1	binary number from 1 to 31 decimal.	Specifies how many print positions the left margin is to be indented for a SYSOUT data set printed on a 3211 Printer with the indexing feature. (JES2 only)
INTRAY	DOINTRAY	003E	1	1-3	binary number from 1 to 255 decimal	Specifies the printer input tray from which to take paper for the print job. Overrides comparable FORMDEF specification.

Table 96. Dynamic Output Text Units and their JCL Equivalents (continued)

JCL Keyword Parameter	*Symbolic key from IEFDOKEY Macro*	*Key in Hex*	*Maximum number of value fields*	*Length of value field*	*Value field*	*Function*
LINDEX	DOLINDEX	0014	1	1	binary number from 1 to 31 decimal.	Specifies how many print positions the right margin is to be moved in from the full page width for a SYSOUT data set printed on a 3211 Printer with the indexing feature. (JES2 only)
LINECT	DOLINECT	0015	1	1	binary number from 0 to 255 decimal	Specifies the maximum lines JES2 is to print on each page. (JES2 only)
MAILBCC	DOMAILBC	0049	32	1-60	EBCDIC text characters	Specifies one or more e-mail addresses of the recipients on the blind copy list.
MAILCC	DOMAILCC	004A	32	1-60	EBCDIC text characters	Specifies one or more e-mail addresses of the recipients on the copy list.
MAILFILE	DOMAILFI	004B	1	1-60	EBCDIC text characters	Specifies the file name of the attachment to an e-mail.
MAILFROM	DOMAILFR	004C	1	1-60	EBCDIC text characters	Specifies the descriptive name or other identifier of the sender of an e-mail.
MAILTO	DOMAILTO	004D	32	1-60	EBCDIC text characters	Specifies one or more e-mail addresses of the e-mail recipients.
MODIFY (module name)	DOMODIF6	0016	1	1-4	alphanumeric or national (@, $, #) characters	Specifies a copy-modification module in SYS1.IMAGELIB to be used by JES to print the data set on a 3800 Printing Subsystem.
MODIFY (trc)	DOMODIF7	0017	1	1	binary number from 0 to 3	Specifies which character arrangement table is to be used. Related to the CHARS key.
NAME	DONAME	002D	1	1-60	EBCDIC text characters	Specifies the preferred name of the owner of the SYSOUT data set.
NOTIFY	DONOTIFY	002F	4	1-17	node (optional) and userid	Sends a print complete message to the specified destination.
OFFSETXB	DOXOFSTB	0043	1-10	1-13	See *z/OS MVS JCL Reference*.	Specifies the X offset of the logical page origin from the physical page origin for the back side of each page. Overrides comparable FORMDEF specification.
OFFSETXF	DOXOFSTF	0041	1-10	1-13	See *z/OS MVS JCL Reference*.	Specifies the X offset of the logical page origin from the physical page origin for the front side of each page. Overrides comparable FORMDEF specification.

Table 96. Dynamic Output Text Units and their JCL Equivalents (continued)

JCL Keyword Parameter	*Symbolic key from IEFDOKEY Macro*	*Key in Hex*	*Maximum number of value fields*	*Length of value field*	*Value field*	*Function*
OFFSETYB	DOYOFSTB	0044	1-10	1-13	See *z/OS MVS JCL Reference*.	Specifies the Y offset of the logical page origin from the physical page origin for the back side of each page. Overrides comparable FORMDEF specification.
OFFSETYF	DOYOFSTF	0042	1-10	1-13	See *z/OS MVS JCL Reference*.	Specifies the Y offset of the logical page origin file the front side of each page. Overrides comparable FORMDEF specification.
OUTBIN	DOOUTBIN	2023	1	4	binary number from 1 to 65535 decimal	Specifies the printer output bin ID.
OUTDISP (normal job completion)	DOOUTDB	002B	1	1	X'80' for WRITE X'40' for HOLD X'20' for KEEP X'10' for LEAVE X'08' for PURGE	Specifies the SYSOUT data set disposition for normal job completion. (JES2 only)
OUTDISP (abnormal job completion)	DOOUTDC	002C	1	1	X'80' for WRITE X'40' for HOLD X'20' for KEEP X'10' for LEAVE X'08' for PURGE	Specifies the SYSOUT data set disposition for abnormal job completion. (JES2 only)
OVERLAYB	DOOVRLYB	0040	1	1-8	alphanumeric or national ($, #, @) characters	Specifies that the named medium overlay is to be placed on the back side of each sheet to be printed. The overlay is printed in addition to overlays specified in the FORMDEF.
OVERLAYF	DOOVRLYF	003F	1	1-8	alphanumeric or national ($, #, @) characters	Specifies that the named medium overlay is to be placed on the front side of each sheet to be printed. The overlay is printed in addition to overlays specified in the FORMDEF.
OVFL	DOOVFL	0033	1	1	X'80' for ON X'40' for OFF	Specifies whether or not JES3 should test for page overflow on an output printer. (JES3 only)
PAGEDEF	DOPAGEDE	001F	1	1-6	alphanumeric or national (@, $, #) characters	Names a library member that PSF uses in printing the SYSOUT data set on a 3800 Printing Subsystem Model 3.
PIMSG	DOPIMSG	0021	2	1	X‘80’ for NO X‘40’ for YES The second value field is a two-byte number from 0 through 999 decimal, having a length field of 2.	Indicates that messages from a functional subsystem should or should not be printed in the listing following the SYSOUT data set. Printing terminates if the number of printing errors exceeds the second value field.

Table 96. Dynamic Output Text Units and their JCL Equivalents (continued)

JCL Keyword Parameter	*Symbolic key from IEFDOKEY Macro*	*Key in Hex*	*Maximum number of value fields*	*Length of value field*	*Value field*	*Function*
PORTNO	DOPORTNO	0045	1	2	binary number from 1 to 65535 decimal	Specifies the TCP port number at which the FSS (for example, Infoprint Server) connects to the printer rather than connecting to LPD on the printer. Specify either PORTNO or PRTQUEUE, but not both. PRTQUEUE indicates the queue used when connecting to LPD on the printer.
PRMODE	DOPRMODE	0018	1	1-8	alphanumeric characters	Identifies the process mode required to print the SYSOUT data set.
PRTATTRS	DOPRTATT	0050	1	1-127	EBCDIC text characters	Specifies an Infoprint Server job attribute. The *z/OS Infoprint Server User's Guide* documents job attribute names and syntax for acceptable values.
PRTOPTNS	DOPROPTN	0039	1	1-16	see *z/OS MVS JCL Reference*.	Named entity that can specify additional print options for FSS use.
PRTERROR	DOPRTERR	003C	1	1	X'80' for QUIT X'40' for HOLD X'20' for DEFAULT	Specifies the action to be taken on a SYSOUT data set while being printed by PSF/MVS for a terminating error.
PRTQUEUE	DOPRTQUE	0038	1	1-127	see *z/OS MVS JCL Reference*.	Identifies the target print queue for use by the FSS.
PRTY	DOPRTY	0019	1	1	binary number from 0 to 255 decimal	Specifies initial priority at which the SYSOUT data set enters the output queue.
REPLYTO	DOREPLYT	004E	1	1-60	EBCDIC text characters	Specifies the e-mail address to which recipients of the e-mail can respond.
RESFMT	DORESFMT	0046	1	1	X'80' for P249 X'40' for P300	Specifies the resolution used to format the print data set.
RETAINF	DORETANF	0037	1	1-10	see *z/OS MVS JCL Reference*.	Specifies the failed transmission retain time for use by the FSS.
RETAINS	DORETANS	0036	1	1-10	see *z/OS MVS JCL Reference*.	Specifies the successful transmission retain time for use by the FSS.
RETRYL	DORETRYL	0035	1	1-3	see *z/OS MVS JCL Reference*.	Specifies the maximum number of transmission retries used by the FSS.
RETRYT	DORETRYT	0034	1	1-8	see *z/OS MVS JCL Reference*.	Specifies the length of time that the FSS will wait between retries.
ROOM	DOROOM	0026	1	1-60	EBCDIC text characters	Specifies the room identification to be associated with the SYSOUT data set.

Table 96. Dynamic Output Text Units and their JCL Equivalents (continued)

JCL Keyword Parameter	*Symbolic key from IEFDOKEY Macro*	*Key in Hex*	*Maximum number of value fields*	*Length of value field*	*Value field*	*Function*
SYSAREA	DOSYSARE	0024	1	1	X‘40’ for YES X‘80’ for NO	Indicates whether you want to use the system printable area of each output page. YES means you want to use the area. NO means you do not want to use the area.
THRESHLD	DOTHRESH	0022	1	4	binary number from 1 to 99999999 decimal	Specifies the maximum size for a sysout data set. Use it to obtain simultaneous printing of large data sets or many data sets from one job. (JES3 only)
TITLE	DOTITLE	002A	1	1-60	EBCDIC text characters	Specifies a title for the SYSOUT data set to be placed on the separator pages.
TRC	DOTRC	001A	1	1	X‘80’ for NO X‘40’ for YES	Specifies whether or not the SYSOUT data set's records contain table reference codes (TRC) as the second character.
UCS	DOUCS	001B	1	1-4	alphanumeric or national (@, $, #) characters	Specifies universal character set, print train, or character arrangement table for a 3800 Printing Subsystem.
USERDATA	DOUSERDA	0031	16	1-60	EBCDIC text '40'X - 'FE'X	User-oriented information as defined by the installation.
USERLIB	DOUSERLI	002E	8	44	cataloged data set name	Specifies the names of libraries containing AFP resources.
USERPATH	DOUSERPAT	004F	8	1–255	SPECIAL text. See *z/OS MVS JCL Reference*.	Specifies up to eight z/OS UNIX file paths containing resources to be used by PSF when processing SYSOUT data sets.
WRITER	DOWRITER	001C	1	1-8	alphanumeric or national (@, $, #) characters	Names an external writer to process the SYSOUT data set rather than JES.

Note:

1. A "length of value field" greater than 4 characters is acceptable without error provided you supply no more than 4 significant characters in that field. In the following example, a length of X'13' is allowed when the 4–character string "CHAR" is coded; the 5–character string "CHARS" will be flagged as an error.

```
0002 0001 0013 CL19'CHAR'    is acceptable
0002 0001 0013 CL19'CHARS'   is not acceptable
```

Deleting output descriptors

To delete an output descriptor, use the OUTDEL macro. The results that you get from issuing the OUTDEL macro depend on which JES you are using. If you use the OUTDEL macro in an address space running under JES2 Version 4.1.0 or later, or JES3 Version 4.2.1 or later, the deleted output descriptor's storage is freed before or at step termination. If, however, you issue the OUTDEL macro in an address space running under a version earlier than JES2 4.1.0, the deleted output descriptor's storage is always freed at the end of the job. The details of these different results follow.

OUTDEL is used in an address space running under JES2 Version 4.1.0 or later, or running under JES3 Version 4.2.1 or later

When you delete an output descriptor, the output descriptor is logically removed from the system. The deleted output descriptor's storage is freed before or at step termination. The system frees the output descriptor's storage once all allocated SYSOUT data sets that refer to the output descriptor have been either opened or, if they are never opened, when they are unallocated. If no SYSOUT data sets refer to the output descriptor, the system frees the storage immediately.

OUTDEL is used in an address space running under an earlier version of JES2 than JES2 4.1.0

As in the case above, when you delete an output descriptor, it is logically removed from the system. However, the output descriptor's storage is always freed at the end of the job. If a program creates too many descriptors, it can run out of storage in the address space. Dynamic output cannot reuse the storage that a deleted descriptor occupies. An output descriptor varies in size, depending on the SYSOUT processing options it contains.

Like OUTADD, the OUTPUT JCL statement also creates an output descriptor. If an output descriptor was created by an OUTPUT JCL statement, you cannot delete it by using the OUTDEL macro. If you try to do this, you get return code 4 with reason code 403 hex.

When a program invokes dynamic allocation to allocate a SYSOUT data set, the data set can refer to one or more output descriptors. Although the descriptors must exist before dynamic allocation is invoked, they do not have to exist afterwards. So, you can delete the descriptor immediately after allocating the data set. Deleting the descriptor does not affect the data set, which can exist even though its descriptor is deleted.

A dynamically allocated SYSOUT data set can refer to either a named or a default output descriptor. The dynamically allocated data set refers to a named output descriptor by using the dynamic allocation key, X'8002' (DALOUTPT). By omitting this key, the dynamically allocated data set refers to a default output descriptor. You define the default output descriptors by using the JCL OUTPUT statement with a DEFAULT=YES parameter, or by having a program invoke dynamic output using the text unit key, X‘000B’(DODEFAUL).

If, after the data set is allocated and its descriptor is deleted, you create another descriptor with the same name but with different text unit characteristics, the characteristics of the data set are not changed. The processing options of the data set depend only on the descriptor that was used at allocation time. Also note that, if you unallocate the data set while its output descriptor exists, the descriptor is not affected.

The syntax of OUTDEL allows only a list form and an execute form. To specify the name of the descriptor that is to be deleted, use the execute form.

Specifying SYSOUT without an output descriptor

When you use the OUTADD macro or an OUTPUT JCL statement to specify the processing options for a SYSOUT data set, the system creates an output descriptor that you can reference when you invoke dynamic allocation, as described in “OUTPUT statement reference - Key = '8002'” on page 651. However, a more convenient method of specifying the processing options for SYSOUT, a method that does not create any output descriptor, is to use dynamic allocation. Dynamic allocation lets you specify, through the use of certain keys, the processing options for the SYSOUT data set that you are allocating. However, this method has limitations:

1. The selection of SYSOUT processing options available through dynamic allocation is very small, compared to the selection available through dynamic output. (See the third column in Table 97.)
2. Because the SYSOUT processing options are not contained in any output descriptor, they cannot be referenced by other invocations of dynamic allocation.
3. Although the options available through dynamic allocation correspond in some cases to the options available through dynamic output, the correspondence is not necessarily equivalent. Make sure that the dynamic allocation option is functionally close enough to the dynamic output option before you decide to use the dynamic allocation processing option.

If you are planning to write a program that uses both dynamic output and dynamic allocation, determine if the SYSOUT processing option that you need is available through dynamic allocation.

For example, the DEST and BURST options of dynamic output are also available through dynamic allocation. If DEST is all you need, you can use the DALUSRID option of dynamic allocation; you do not need dynamic output. If you need BURST, it may be easier to use the DALBURST option of dynamic allocation; again, you do not need dynamic output. Thus, if the options that you need are available through dynamic allocation, you can avoid coding the OUTADD macro. The following table lists the dynamic output SYSOUT processing options (column two) that are also available through dynamic allocation (column three):

Table 97. Alternate Sources of SYSOUT Processing Options. Comparable SYSOUT processing options in OUTPUT JCL, dynamic output, dynamic allocation, and DD JCL

OUTPUT JCL Parameter	*Dynamic Output Key, equivalent to OUTPUT JCL parameter*	*Dynamic Allocation Key, comparable to dynamic output key*	*DD JCL Parameter, equivalent to dynamic allocation key*
BURST	DOBURST	DALBURST (0064)	BURST
CHARS	DOCHARS	DALCHARS (0065)	CHARS
CLASS	DOCLASS	DALSYSOU (0018)	SYSOUT (class)
COPIES (data set count)	DOCOPIE9	DALCOPYS (001D)	COPIES (data set count)
COPIES (group value count)	DOCOPIEA	DALCOPYG (0066)	COPIES (group value count)
DEST	DODEST	DALUSRID (0063), or DALSUSER (0058)	DEST
FCB	DOFCB	DALFCBIM (0025)	FCB
FLASH (overlay name)	DOFLASE	DALFFORM (0067)	FLASH (overlay name)
FLASH (count)	DOFLASF	DALFCNT (0068)	FLASH (count)
FORMS	DOFORMS	DALSFMNO (001A)	SYSOUT (form name)

Table 97. Alternate Sources of SYSOUT Processing Options (continued). Comparable SYSOUT processing options in OUTPUT JCL, dynamic output, dynamic allocation, and DD JCL

OUTPUT JCL Parameter	*Dynamic Output Key, equivalent to OUTPUT JCL parameter*	*Dynamic Allocation Key, comparable to dynamic output key*	*DD JCL Parameter, equivalent to dynamic allocation key*
MODIFY (module name)	DOMODIF6	DALMMOD (0069)	MODIFY (module name)
MODIFY (trc)	DOMODIF7	DALMTRC (006A)	MODIFY (trc)
OUTDISP (normal output disposition)	DOOUTDB	DALSHOLD (0059)	HOLD
OUTDISP (abnormal output disposition)	DOOUTDC	DALSHOLD (0059)	HOLD
UCS	DOUCS	DALUCS (0029)	UCS
WRITER	DOWRITER	DALSPGNM (0019)	SYSOUT (writer name)

Dynamic output programming example

The following example uses dynamic output to create an output descriptor and allocates a SYSOUT data set that references the descriptor. Then it deletes the descriptor. The characteristics of the descriptor are specified in the program's input, except for BURST, which is hard-coded.

```
DYN      CSECT                        ENTRY LINKAGE
         USING *,15
         STM   14,12,12(13)
         BALR  12,0
BEGIN    DS    0H
         USING BEGIN,12
*
* REG 1 POINTS TO TWO INPUT WORDS: WORD 1 POINTS TO
* REQUESTED SYSOUT OPTIONS.  WORD 2 POINTS TO OUTPUT
* AREA FOR RETURN AND REASON CODES.
*
         LA    4,OUTPUT_DATA          ADR OF OUTPUT DATA
         ST    4,4(0,1)               PUT ADR IN PARM LIST
         L     4,0(0,1)               POINTER TO INPUT
*
* MOVE INPUT DATA INTO THE TEXT UNITS
*
         USING INPUT_DATA,4           MAP INPUT DATA
         MVC   COPYPRM,COPYIN         HOW MANY COPIES
         MVC   GROUP#1,GROUPIN1       FIRST GROUP VAL
         MVC   GROUP#2,GROUPIN2       SECOND GROUP VAL
         MVC   FORMDPRM,FORMDIN       WHAT KIND OF FORM
         OC    POINTER4,ENDMASK       HIGH ORDER BIT ON
*                                     INDICATES END OF
*                                     TEXT UNIT POINTER
*                                     LIST
*
* ISSUE OUTADD TO CREATE AN OUTPUT DESCRIPTOR
*
         OUTADD NAME=ODNAME,TEXTPTR=TXTPTRS1,        C
               MF=(E,ADDPARML)
         ST    15,OUTADD_RETCODE      SAVE RETURN CODE
         ST    0,OUTADD_RESCODE       SAVE REASON CODE
         ST    1,OUTADD_KEY           SAVE KEY IN ERROR
         LTR   15,15                  OUTADD RETCODE
         BNZ   SKIP                   BRANCH IF BAD
*
* INVOKE DYNALLOC TO ALLOCATE THE SYSOUT DATA SET THAT
* GOES WITH THE OUTPUT DESCRIPTOR OUT1.
*
         B     DYNAL                  DYNALLOC ROUTINE
*
* INVOKE OUTDEL TO DELETE THE DESCRIPTOR, THEN EXIT
*
DELETE   OUTDEL NAME=ODNAME,MF=(E,DELPARML)
         ST    15,OUTDEL_RETCODE      SAVE RETURN CODE
         ST    0,OUTDEL_RESCODE       SAVE REASON CODE
SKIP     DS    0H                     HERE IF CREATE BAD
         LM    14,12,12(13)           RETURN TO CALLER
         SLR   15,15
         BR    14
```

Dynamic output programming example (continued)

The text unit pointer list references four text units. The first is hard coded and the other three are set up dynamically.

```
* POINTERS TO FOUR TEXT UNITS.
*
TXTPTRS1 DS 0F               TEXT UNIT POINTERS
POINTER1 DC AL4(BURST)       ADR TU FOR BURST
POINTER2 DC AL4(COPIES)      ADR TU FOR COPIES
POINTER3 DC AL4(COPYGRP)     ADR TU FOR GROUPING
POINTER4 DC AL4(FORMDEF)     ADR TU FOR FORMDEF
*
BURST    DS 0H               BURST SELECTION TU
         DC AL2(DOBURST)     KEY=BURST
         DC XL2'0001'        HOW MANY VAL FIELDS
         DC XL2'0001'        LEN OF VALUE FIELD
         DC XL1'02'          VAL FLD (HARD CODED BURST=Y)
*
COPIES   DS 0H               NBR COPIES TU
         DC AL2(DOCOPIE9)    KEY=COPIES
         DC XL2'0001'        HOW MANY VAL FIELDS
         DC XL2'0001'        LEN OF VALUE FIELD
COPYPRM  DS XL1              VALUE FIELD (TO BE ALTERED)
*
COPYGRP  DS 0H               COPY GROUPING TU
         DC AL2(DOCOPIEA)    KEY=COPIES(GROUP)
         DC XL2'0002'        HOW MANY VAL FIELDS
         DC XL2'0001'        LEN OF VALUE FIELD 1
GROUP#1  DS XL1              VALUE FIELD 1 (TO BE ALTERED)
         DC XL2'0001'        LEN OF VALUE FIELD 2
GROUP#2  DS XL1              VALUE FIELD 2 (TO BE ALTERED)
*
FORMDEF  DS 0H               FORMS DEFINITION TU
         DC AL2(DOFORMD)     KEY=FORMDEF
         DC XL2'0001'        HOW MANY VAL FIELDS
         DC XL2'0006'        LEN OF VALUE FIELD
FORMDPRM DS CL6              VALUE FIELD (TO BE ALTERED)
*
* CONSTANTS FOR DYNAMIC OUTPUT INVOCATION
*
ODNAME   DC CL8'OUT1    '    OUTPUT DES NAME
ENDMASK  DC X'80000000'      HIGH ORDER BIT ON
*
* PARAMETER LISTS FOR OUTADD AND OUTDEL
*
ADDPARML OUTADD MF=L
DELPARML OUTDEL MF=L
         IEFDOKEY            DEFINES DOBURST, ETC
*
* Program returns ptr to this table to the caller
*
OUTPUT_DATA    DS     0XL28      INFORMATION RETURNED
OUTADD_RETCODE DS     XL4'0'     OUTADD RETURN CODE
OUTADD_RESCODE DS     XL4'0'     OUTADD REASON CODE
OUTADD_KEY     DS     XL4'0'     OUTADD KEY IN ERROR
OUTDEL_RETCODE DS     XL4'0'     OUTDEL RETURN CODE
OUTDEL_RESCODE DS     XL4'0'     OUTDEL REASON CODE
DYNAL_RETCODE  DS     XL4'0'     DYNALLOC RETURN CODE
DYNAL_REASON   DS     XL4'0'     DYNALLOC REASON CODES
```

Dynamic output programming example (continued)

This is the subroutine that dynamically allocates the SYSOUT data set that references the output descriptor.

```
* INVOKE DYNALLOC TO ALLOCATE A SYSOUT DATA SET.  SPECIFY
* SYSOUT CLASS A AND REFERENCE THE OUTPUT DESCRIPTOR
* 'OUT1'.  SET UP ADDRESSABILITY TO THE DYNAMIC ALLOCATION
* (SVC 99) REQUEST BLOCK.
*
DYNAL    DS    0H
         LA    10,SVC99PRM         PTR TO SVC 99 RB PTR
         LA    11,SVC99PRM+4       PTR TO SVC 99 RB
         USING S99RBP,10           RB PTR
         USING S99RB,11            RB
         ST    11,S99RBPTR         SET RB PTR TO RB
         OI    S99RBPTR,S99RBPND   PTR HIGH ORDER BIT ON
*
* INITIALIZE THE SVC 99 REQUEST BLOCK
*
         XC    S99RB(RBLEN),S99RB  ZERO THE RB
         MVI   S99RBLN,RBLEN       RB LENGTH
         MVI   S99VERB,S99VRBAL    RB VERB CODE=ALLOC
         LA    1,TXTPTRS2          ADR SVC 99 TEXT PTRS
         ST    1,S99TXTPP          STORED IN RB
         OI    POINTERB,S99TUPLN   HIGH ORDER BIT ON
*                                  INDICATES LAST TEXT
*                                  UNIT POINTER
         LR    1,10                ADR OF RB POINTER
*
* ISSUE DYNALLOC TO ALLOCATE A SYSOUT DATA SET THAT
* REFERENCES THE OUTPUT DESCRIPTOR, OUT1
*
         DYNALLOC                  INVOKES SVC99
         ST    15,DYNAL_RETCODE    SAVE RETURN CODE
         MVC   DYNAL_REASON,S99RSC SAVE REASON CODE
         B     DELETE              TO DEL THE OUT DES
*
* DECLARE SPACE FOR THE RB, WHICH IS MAPPED BY IEFZB4D0
*
         DS 0F
SVC99PRM DS    CL(RBLEN+4)          RB PTR & RB STORAGE
RBLEN    EQU   (S99RBEND-S99RB)     LENGTH OF RB
```

Dynamic output programming example (concluded)

The SVC 99 text units and request block are shown. Definitions for DALOUTPT and DALSYSOU, which are mapped by IEFZB4D2, are not shown. See IEFZB4D2 in *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/).

```
 * POINTERS TO THE TWO DYNAMIC ALLOCATION TEXT UNITS
 *
 TXTPTRS2 DS 0F
 POINTERA DC AL4(SYSOUT)            PTR TO SYSOUT TU
 POINTERB DC AL4(OUTPUT)            PTR TO OUTPUT TU
 *
 SYSOUT   DS 0H                     SYSOUT TU
          DC AL2(DALSYSOU)          KEY = SYSOUT
          DC XL2'0001'              HOW MANY VAL FIELDS
          DC XL2'0001'              LEN OF VALUE FIELD
          DC CL1'A'                 VAL FIELD, SYSOUT=A
 *
 OUTPUT   DS 0H                     DATA SET NAME TU
          DC AL2(DALOUTPT)          KEY=OUTPUT REFERENCE
          DC XL2'0001'              HOW MANY VAL FIELDS
          DC XL2'0008'              LEN OF VALUE FIELD
          DC CL8'OUT1'              VAL FLD, OUTPUT=*.OUT1
 *
 * SVC 99 REQUEST BLOCK DSECT, ABRIDGED EXPANSION
 *
          IEFZB4D0       MAP SVC 99 REQUEST BLOCK
+S99RBP   DSECT
+S99RBPTR DS     F       REQUEST BLOCK POINTER
+S99RB    DSECT          REQUEST BLOCK
          DS     0F
+S99RBLN  DS     CL1     LENGTH OF REQUEST BLOCK
+S99VERB  DS     CL1     VERB CODE
+S99VRBAL EQU    X'01'   ALLOCATION
+S99FLAG1 DS     0CL2    FLAGS
+S99FLG11 DS     CL1     FIRST FLAGS BYTE
+S99FLG12 DS     CL1     SECOND BYTE OF FLAGS
+S99RSC   DS     0CL4    REASON CODE FIELDS
+S99ERROR DS     XL2     ERROR REASON CODE
+S99INFO  DS     XL2     INFORMATION REASON CODE
+S99TXTPP DS     F       ADDR OF LIST OF TEXT UNIT PTRS
+S99S99X  DS     F       ADDR OF REQ BLK EXTENSION
+S99FLAG2 DS     0CL4    FLAGS FOR AUTHORIZED FUNCTIONS
+S99FLG21 DS     CL1     FIRST BYTE OF FLAGS
+S99FLG22 DS     CL1     SECOND BYTE OF FLAGS
+S99FLG23 DS     CL1     THIRD BYTE OF FLAGS
+S99FLG24 DS     CL1     FOURTH BYTE OF FLAGS
+S99RBEND EQU    *       END MARKER
          IEFZB4D2       MAP SVC 99 KEY TABLE
 *
 * FIELDS INDICATING PRINTED OUTPUT CHARACTERISTICS
 *
 INPUT_DATA DSECT                   INPUT MAPPING
 COPYIN     DS XL1                  HOW MANY COPIES
 GROUPIN1   DS XL1                  COPIES GROUP VAL
 GROUPIN2   DS XL1                  COPIES GROUP VAL
 FORMDIN    DS CL6                  WHAT KIND OF FORMS
          END
```

Chapter 28. Scheduler JCL facility (SJF)

Scheduler JCL facility (SJF) services assist an application in processing the data definition (DD) and system output (SYSOUT) attributes that can be specified on JCL or through dynamic output. SJF services can be used by an application that dynamically allocates SYSOUT data sets.

The SJFREQ macro allows an application to call the following SJF services:

SJFREQ RETRIEVE
Retrieves keyword subparameter information in text unit format from output descriptors. These output descriptors can be specified either on an OUTPUT JCL statement or through dynamic output.

SJFREQ SWBTU_MERGE
Updates SYSOUT characteristics by merging two scheduler work block text unit (SWBTU) lists.

SJFREQ VERIFY
Validates a statement name, operands, and subparameters, and builds text units that can be used as input for the dynamic output service.

SJFREQ TERMINATE
Cleans up the SJF environment established by an SJF service.

This information discusses SJF services and their use for an application program. The descriptions of the individual services identify the types of applications likely to use the services and the circumstances under which an application uses them.

Another SJF macro, SWBTUREQ, allows applications to obtain information from scheduler work block text units (SWBTUs). The SWBTUREQ macro and the SJFREQ macro are described in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

The table that follows outlines the contents of the information.

Topic	**Content**
"Understanding SJF terms" on page 740	Defines terms used in the discussion of SJF services.
"The SJF environment" on page 741	Describes the SJF environment and considerations for reusing the environment.
"Retrieving output descriptor information (SJFREQ macro with RETRIEVE)" on page 741	Describes the SJFREQ RETRIEVE service.
"Merging SWBTUs (SJFREQ macro with SWBTU_MERGE)" on page 742	Describes the SJFREQ SWBTU_MERGE service.
"Validating and building text units (SJFREQ macro with VERIFY)" on page 743	Describes the SJFREQ VERIFY service.
"Freeing the SJF environment (SJFREQ macro with TERMINATE)" on page 750	Describes the SJFREQ TERMINATE service.
"Understanding the OUTDES statement" on page 751	Describes the OUTDES statement and considerations for using it.

Understanding SJF terms

To use the SJF macros and services, you must understand several terms. These terms are defined in this information. The statement below illustrates some of the definitions that follow. The definitions refer to specific portions of the statement.

```
STMNT OPTIONS(A,B,(c,d,e)) TYPE
```

operand
A stand-alone word specified on a statement. It has no sub-values specified with it. In the statement example, **TYPE** is an operand.

keyword operand
A word specified on a statement that has a value or list of values (subparameter) specified with it. In the statement example, **OPTIONS** is a keyword operand.

subparameter
A value or list of values specified with a keyword operand. In the statement example, **A** and **B** and **(c,d,e)** are all subparameters.

subparameter number
The position of a subparameter within a keyword operand. In the statement example, **A** is subparameter number 1, **(c,d,e)** is subparameter number 3.

sublist element number
The position of a sublist element within a sublist. In the statement example, **d** is sublist element number 2.

sublist
A single subparameter that consists of a list of values. In the statement example, **(c,d,e)** is a sublist.

sublist element
One value within a sublist. In the statement example, **c** and **d** and **e** are each sublist elements.

choice The data type of a predefined set of subparameters defined for a keyword operand. Only one subparameter may be selected and specified. In the following example, only one of the four choices in parentheses may be specified.

```
DATACK(BLOCK|UNBLOCK|BLKCHAR|BLKPOS)
```

character
The data type of a string of characters. Allowable characters and string length are defined for each keyword. For some keywords the string can be repeated, delineated by periods. Each delineated string is called a level. In the example **DEST(POK.POST)**, POK and POST are levels.

numeric
The data type of a character representation of an integer, for example, **2** in COPIES(2).

text unit
Parameter information that represents a keyword operand and its subparameters. More than one text unit may be needed to represent an entire keyword operand. For more information about text units, see Chapter 27, “Dynamic output,” on page 719.

text unit pointer list
A contiguous list of 4-byte pointers. Each pointer points to a specific text unit. The last pointer in the list must have the left-most (high-order) bit set on.

output descriptor
An object that represents the output format characteristics of a SYSOUT data set.

OUTDES statement
An alternate form of OUTPUT JCL statements that is used in dynamic output environments. (See "Understanding the OUTDES statement" on page 751 for the syntax and other information about OUTDES.)

verb Identification information that appears in the SWBTU prefix.

The SJF environment

The SJF environment consists of a work area and an ESTAE-type recovery routine. Applications can request that SJFREQ reuse the environment for multiple calls of the same service.

When multiple calls are required to complete the VERIFY function, an application must request that the environment be reused. To request SJFREQ to reuse the environment, the application must use the same value in SJVESTOR field of the VERIFY parameter list (IEFSJVEP) that was returned on the previous call.

Because an SJF environment can be reused over many calls, the application must indicate on the last SJFREQ call that the environment should be freed, or the application can issue SJFREQ TERMINATE to free the environment.

The system establishes an ESTAE-type recovery routine to release the work area in the event of an abnormal termination. The recovery routine is in effect for the duration of the SJF call. If an application reuses the work area, the recovery routine for the work area is still in effect even after the system returns control to the application.

When the application reuses the SJF environment, the SJF recovery routine protects the environment on behalf of the application. The SJF recovery routine gets control under the following conditions:

- If an abnormal termination occurs while the application is processing and while SJF's environment still exists, then:
  - Abnormal termination continues
  - The application cannot reuse the environment that was referenced by SJVESTOR on input for the call that abnormally terminated.
- If the abnormal termination occurs while SJF is processing and the SJF environment still exists, the application:
  - Receives a hexadecimal return code 14 from the SJFREQ macro, indicating an abnormal termination occurred
  - Cannot reuse the environment that was referenced by SJVESTOR on input for the call that abnormally terminated.

Retrieving output descriptor information (SJFREQ macro with RETRIEVE)

Applications can use the SJFREQ RETRIEVE service to retrieve keyword subparameter information in text unit format from output descriptors. These output descriptors can be specified either on an OUTPUT JCL statement or through dynamic output. The application invokes this service to retrieve output descriptor information in a functional subsystem environment. The following topics describe the tasks an application must perform to invoke the RETRIEVE service.

Initializing the keyword list

The application needs to provide a keyword list (SJRELIST) to the RETRIEVE service. The keyword list contains paired fields; each pair consists of a keyword field and a pointer field. In the list, the application specifies the JCL keywords for which information is to be retrieved. For each keyword specified, the RETRIEVE service returns a pointer to the text unit pointer list associated with the keyword.

The SJFREQ macro description in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* shows the SJRELIST paired fields and their offsets and lengths. The description indicates the fields that the application must initialize.

Establishing a storage area

For each RETRIEVE request, the application needs to establish a storage area in which SJF is to return the output descriptor information. The size of this storage area depends on the number of keywords for which the application requests information.

The application specifies the address and size of this storage area in the SJF RETRIEVE parameter list. *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* describes the parameter list.

Information returned from SJF RETRIEVE processing

On return from RETRIEVE processing, the keyword list (SJRELIST) contains paired fields, each pair consisting of a JCL keyword and a pointer to the text unit pointer list for that keyword.

The application-provided storage area contains the text unit pointers list and the individual text units associated with each keyword.

The text units associated with a particular keyword can contain one or more keys. If the application does not specify data corresponding to a particular key, that key might not be returned or a key might be returned with a parameter count of zero. The application must interpret each returned key to determine whether it contains zero or some other parameter count value.

Merging SWBTUs (SJFREQ macro with SWBTU_MERGE)

Applications sometimes need to update the output characteristics associated with a system output (SYSOUT) data set. The SJFREQ SWBTU_MERGE service allows an application to update these SYSOUT characteristics by merging two SWBTU lists. The application can request that a base SWBTU list be merged with a merge SWBTU list. A base SWBTU list contains the current data set characteristics; a merge SWBTU list contains additional data set characteristics or the base list with changes. The SJFREQ SWBTU_MERGE service also allows an application to remove SWBTUs from a base SWBTU list.

An application can request the SWBTU_MERGE service to allow certain errors during input processing by setting the SJSMWARN bit in the parameter list. Setting the SJSWARN bit allows errors for which the return code is 0, and the reason code is X'0CA' through X'243'. Use the SJSMWARN bit only when you know the keywords you are using are not compatible with your current system release. Also note that even though you may have more than one error in a SWBTU list, \JSMEKER and SJSMMKER contain information about only one error upon return from an invocation. You might need to repeat this process of fixing an error and invoking the service several times before eliminating all the errors. See *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for information about filling in the parameter list.

Use caution when setting the SJSMWARN bit. When SJSMWARN is on, errors are allowed for any key or parameter, not just for the keys or parameters for which you set SJSMWARN on (keys that are not compatible with your current system release).

Validating and building text units (SJFREQ macro with VERIFY)

An application's processing may involve the dynamic allocation of SYSOUT data sets. As part of this processing, the application allows the user (a programmer or another program) to specify the output format characteristics of the SYSOUT data set. These characteristics are referred to as output descriptor information. The application needs to process the output descriptor information and pass the information to the dynamic output service. Dynamic output requires input in text unit format.

SJFREQ VERIFY validates the output descriptor information specified by the user or application and returns data in text unit format. A text unit pointer list, created by VERIFY, can be used as input to dynamic output. The application then has the data in the format needed by dynamic output.

An application that provides a menu through which users enter print options might use SJF to check the user-specified options. For example, an application might have a menu that allows users to indicate paper size, number of copies, number of lines per page, font, a delivery address, and a user name. After a user enters this information, the application can call SJF to validate and process the options so the system can use the options to create the user's output.

SJFREQ VERIFY functions

VERIFY validates the statement name, operands, keyword operands, and subparameters that make up the OUTDES statement, and builds text units for valid statement information. “Understanding the OUTDES statement” on page 751 describes the OUTDES statement syntax and considerations for using the statement. Use the VERIFY service to validate statement information by filling in the required fields of the VERIFY parameter list (IEFSJVEP) for the function your application performs. *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* defines each parameter list field and summarizes the fields needed for each of the SJFREQ VERIFY functions. VERIFY performs several functions that an application can use:

- An application can validate a statement name, or keyword operand, or operand prior to processing the values of subparameters or sublist elements. To do this prior validation, the application can provide the following in the VERIFY parameter list:
  - The operand or keyword operand. VERIFY determines that the operand or keyword operand is defined to the statement, and that the statement name is OUTDES.
  - The statement name. VERIFY determines that the statement being processed is OUTDES.

- An application can only process a statement name, keyword operand, and either a single subparameter or a single sublist element on the same VERIFY call. To do this statement validation and begin building a text unit, an application must provide the statement name, keyword operand, and the subparameter or sublist element on the same call. VERIFY determines if the subparameter or sublist element value is defined to the keyword operand and the OUTDES statement and if the values are valid. If the values are valid, VERIFY begins or continues to build a text unit for the subparameter or sublist element.

  To build a text unit for an entire keyword operand, pass each subparameter and sublist element for the keyword operand to VERIFY on separate calls. For example, in the statement that follows, VERIFY must be called five times, once for each subparameter or sublist element.

  ```
  OPTIONS(A,B,(c,d,e))
  ```

  To build a text unit for an operand, call VERIFY only once.

  Call VERIFY until you finish processing a single OUTDES statement, then use the text units as input to dynamic output before moving on to the next statement.

Preparing to use VERIFY for validating and building text units

An application programmer should consider several things in order to code an application correctly when using VERIFY. These items are listed here and described in the following paragraphs.

- Error Information
- Parameter Delimiters
- Sequence of Parameter Validation
- Operand Processing
- Parameter Overrides
- Last Call Processing
- Using the SJF Environment
- Text Unit Output Area
- Prefixing Data Set Names

Error information: When VERIFY is passed a subparameter or sublist element, VERIFY validates the syntax. If the subparameter or sublist element is correct, VERIFY builds a text unit to represent it. If the subparameter or sublist element is not valid, VERIFY returns information indicating the data was not valid. The reason code identifies the type of error. For some errors VERIFY returns error message text and an operand description. The application can use the text and description to inform the user how to correct the error. See *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for the message text and operand descriptions.

Parameter delimiters: VERIFY does not process the parentheses or commas that might appear in the input as delimiters. The application must parse the statement and identify the input VERIFY is to process. Input passed to VERIFY must not include enclosing quotation marks. If the input is enclosed in quotation marks and contains two consecutive single quotation marks, the application must:

- Convert the consecutive single quotation marks to a single quotation mark
- Remove the enclosing quotation marks
- Pass the input to VERIFY.

For example, pass the string 'BOARD''S' as BOARD'S to VERIFY as the value to process.

VERIFY provides two functions whose use is determined by whether the input value is delimited by single quotation marks. When an input value is delimited by single quotation marks and an application sets parameter field SJVEQUOT on (sets it to one), VERIFY does additional validation. VERIFY determines if the input value may be specified within quotation marks on the statement. If an application does not want to use the quotation mark validation feature, it should set SJVEQUOT to zero.

When an input value is not delimited by single quotation marks and an application specifies a value in SJVEPRFX, VERIFY performs a function called prefixing.

Sequence of parameter validation: For a keyword operand, an application must call VERIFY for each subparameter and sublist element. Call VERIFY with the subparameters and sublist elements in the order in which they appear following the keyword operand. For example, in “Validate and build a text unit for a keyword operand and subparameters” on page 748 the application calls VERIFY for the subparameter GT10, and then calls VERIFY again to process the subparameter GB10. The application must call VERIFY once for each subparameter.

If an application passes VERIFY a subparameter or sublist element that is not valid, VERIFY must be called again for the same subparameter or sublist element until the application does one of the following:
- Passes a valid subparameter or sublist element
- Passes a new operand or keyword operand value
- Requests last call
- Issues a SJFREQ TERMINATE request.

In the case of a subparameter or sublist element that is not correct, the application could prompt its user for a valid value. If the application is unable to produce a valid value, VERIFY processes no more subparameters or sublist elements for the keyword operand; VERIFY builds text units for the subparameters and sublist elements previously processed for the keyword operand. If this is the first subparameter and the first sublist element, and the caller cannot produce a valid value, VERIFY builds no text units for the keyword operand.

Operand processing: Operands are stand-alone words specified on a statement. However, if an application wants a text unit built for an operand, it must pass VERIFY a null subparameter with the operand. The subparameter must be specified as follows:
- 1 as the subparameter number
- 0 as the subparameter length.

If an application does not specify an operand with a subparameter, VERIFY validates the operand, but does not build a text unit.

Parameter overrides: Operands or keyword operands may be processed in any order. When an application specifies the same keyword operand more than once, the text unit represents the last value VERIFY validated for the keyword operand. For example, an application issues the statement below. Assume that the application requests VERIFY to process **NOTIFY(ALB18.POST)** after **NOTIFY(NYS24.BENCH,NYS24.PARKER)**

```
OUTDES out1 NOTIFY(NYS24.BENCH,NYS24.PARKER) NOTIFY(ALB18.POST)
```

When VERIFY completes processing of the statement, the text unit represents **(ALB18.POST)** as the value for the NOTIFY keyword operand.

Last call processing: When the application has processed all the statement information, it must call VERIFY using the last call indicator. See “Requesting last call” on page 750 for an example that specifies last call. This call signifies that no more operands or keyword operands are to be processed for the current statement. After this call, the text units and text unit pointer list for the current statement are available for the application to use. Also, the SJF environment has been refreshed so VERIFY can process another statement with the same environment. (SJVENOCU must have been set on to save the environment.) To invoke VERIFY and request last call, use the parameter list returned from the previous call, and set SJVELSTC on. Setting SJVELSTC on and calling VERIFY makes the text units available for use. This is an additional call after the last subparameter or sublist element is processed. No text units are built on this call.

Using the SJF environment: If an application requests VERIFY to build text units, the application must reuse the SJF environment until a complete statement is verified. So the application must supply the same value in SJVESTOR on each call after the first. VERIFY builds the text units in the area addressed by SJVESTOR. If the application wants to use the same environment to process another OUTDES statement with VERIFY, the application must indicate last call (set SJVELSTC) when VERIFY completes processing the first statement. After the application indicates last call, the application can call VERIFY to process another statement.

Text unit output area: When calling VERIFY to validate information and build text units, the application must pass VERIFY a pointer to an output area to hold the text units. This output area is referred to as the SJF VERIFY text unit output area. Pass the pointer to the output area in SJVETUBP and the length of the output area in SJVETUBL.

An application using the SJFREQ VERIFY service to validate multiple subparameters on one keyword operand must make sure to specify the same SJF VERIFY text unit output area in parameter list field SJVETUBP for each call.

The length of the VERIFY text unit output area must be at least 256 bytes plus area needed to contain the text units built for the statement. A work area of 1K is large enough for any set of text units.

If an application needs to provide additional work areas to contain the text units, the additional work areas can be of any length that will contain at least the text unit VERIFY builds. If more space is needed in the VERIFY text unit output area, VERIFY returns return code 4 and reason code 4B2, indicating the problem. The application must:

- Obtain more storage and call VERIFY again
- Pass the new pointer (SJVETUBP) and text unit output area length (SJVETUBL)
- Pass the subparameter or sublist element that was not processed in the last request.

When an application wants to use the same output area to process multiple keyword operands, the application can request that VERIFY return in parameter field SJVETUBS the amount of output area used by doing the following:

- Set SJVERSBS on.
- Pass the same value in parameter field SJVETUBP as on the initial call for this statement. This value is the pointer to the text unit output area.

If the application uses VERIFY to validate the same statement and specifies a different value in SJVETUBP and SJVERSBS is on, the reason code SJRCIVBD is set. The application specified that the same output area be used on multiple calls by setting SJVERSBS on, but proceeded to specify another output area to be used.

Prefixing data set names: Some keyword operands require a data set name as a subparameter. Data set names may be fully qualified (delimited by single quotation marks) or unqualified (not delimited by single quotation marks). For unqualified data set names, VERIFY allows an application to indicate a prefix in parameter field SJVEPRFX. VERIFY uses this prefix as the first qualifier for the unqualified data set name on the keyword operands for which VERIFY allows prefixing. To indicate a prefix, specify a value in SJVEPRFX and set SJVEQUOT to zero.

Examples of using SJFREQ VERIFY functions

The examples that follow are sample SJFREQ VERIFY calls for processing a statement, operand or keyword operand, subparameter, and sublist element. The calls process the OUTDES statement shown in Figure 74. (See “Understanding the OUTDES statement” on page 751 for more detail on OUTDES.) The calls are written in pseudocode and show the proper sequence of validation. The application has not previously called VERIFY.

```
OUTDES  out1   CHARS(GT10,GB10)  COPIES(1,(2,4,5))
```

Figure 74. Sample OUTDES Statement for Which an Application Wants Text Units

Validate a statement name

The application wants to validate the statement name, OUTDES, prior to validating, parsing, and processing the remainder of the statement. The application can do the following:

- Clear the SJF Verify parameter list SJVEP (set to binary zeros).
- Set the id field, SJVEID, to 'SJVE'.
- Set the parameter list version number to SJVECVER.
- Set the parameter list length, SJVELEN, to SJVELGTH.
- Set the no-cleanup bit, SJVENOCU, on. Save the SJF environment across calls.
- Set the unauthorized-caller bit, SJVEUNAU, on if appropriate for your application.
- Set field SJVEJDVT to zeros.
- Set the statement name field, SJVECMND, to 'OUTDES ' from the statement above.
- Issue SJFREQ REQUEST=VERIFY,PARM=SJVEP.

VERIFY returns with a return and reason code of zero.

Validate a keyword operand

The application now chooses to have VERIFY validate that CHARS is an OUTDES keyword operand. (In this example, and the other examples that follow in this information, pointers below the statement are used to indicate values that are processed in the example.) The application sets OPER_PTR to point to the first byte of CHARS, and determines the length of the CHARS operand. The application issues the VERIFY macro to validate CHARS. The pointers are as follows:

```
OUTDES out1  CHARS(GT10,GB10) COPIES(1,(2,4,5))
             |      |
             |      PARM_PTR
              OPER_PTR
```

Figure 75. OUTDES Statement with Pointers Indicating Values To Be Processed

- Set the operand pointer field, SJVEOPEP, to OPER_PTR
- Set the operand length field, SJVEOPEL, to 5 (length of 'CHARS')
- Issue SJFREQ REQUEST=VERIFY,PARM=SJVEP

VERIFY returns with a return and reason code of zero. This indicates that VERIFY understands the statement and that CHARS is a valid keyword operand for the OUTDES statement.

The application can proceed to validate the keyword operand COPIES or it can request that VERIFY begin building a text unit for the keyword operand, CHARS, and the rest of the OUTDES statement. The application can continue to use the SJF environment established on the VERIFY invocation in this example. To continue to use the environment, the application must save the value in the SJVESTOR field when the SJVEP area is cleared.

Validate and build a text unit for a keyword operand and subparameters

After the application validates the keyword operand CHARS, it calls VERIFY to process the CHARS keyword operand with the subparameter GT10. The pointers are set as in Figure 75.

- Obtain 1K of storage for the SJF Verify text unit output area.
- Clear the SJF Verify parameter list SJVEP (set to zeros).
- Set the id field, SJVEID, to 'SJVE'.
- Set the parameter list version number to SJVECVER.
- Set the parameter list length equal to the length of SJVELGTH.
- Set the no-cleanup bit, SJVENOCU, on. Continue to save the SJF environment across calls.
- Set the unauthorized-caller bit, SJVEUNAU, on if appropriate for your application.
- Set the statement name field, SJVECMND, to 'OUTDES ' from the statement above.
- Set the operand pointer field, SJVEOPEP, to OPER_PTR.
- Set the operand length field, SJVEOPEL, to 5.
- Set the subparameter number field, SJVEPARM, to 1 to indicate first subparameter being processed.
- Set the subparameter pointer field, SJVEPRMP, to PARM_PTR.
- Set the subparameter length field, SJVEPRML, to 4 (length of 'GT10')

- Set the VERIFY text unit output area length field, SJVETUBL, to 1024 (1K obtained above in this example).
- Set the VERIFY text unit output area pointer field, SJVETUBP, to the address of the obtained storage.
- Issue SJFREQ REQUEST=VERIFY,PARM=SJVEP.

VERIFY returns with a return and reason code of zero. The application updates PARM_PTR to point to GB10. The adjusted pointers are as follows:

```
OUTDES out1  CHARS(GT10,GB10) COPIES(1,(2,4,5))
             |          |
             |          PARM_PTR
             OPER_PTR
```

Figure 76. OUTDES Statement with New Pointers Indicating Values To Be Processed

The IEFSJVEP parameter list still contains the results from the previous VERIFY call. Call VERIFY again to validate subparameter GB10 and build a text unit in the text unit output area for the entire CHARS keyword operand. Note that we are still processing the same keyword operand, so the value in SJVETUBP in this example contains the same value passed for all previous subparameters of the keyword operand, CHARS.

- Set the subparameter number field, SJVEPARM, to SJVEPARM +1 (which is now 2).
- Set the subparameter pointer field, SJVEPRMP, to PARM_PTR.
- Set the subparameter length field, SJVEPRML, to 4 (length of 'GB10').
- Issue SJFREQ REQUEST=VERIFY,PARM=SJVEP.

VERIFY returns with a return and reason code of zero.

Validate and build a text unit for a keyword operand with a sublist

After validating and building a text unit for the CHARS subparameters, the application continues by using VERIFY to validate and build text units for COPIES(1,(2,4,5)), the next keyword operand and its subparameters. The second subparameter is a sublist.

The application previously called VERIFY to validate and build the text unit for the CHARS keyword operand as indicated above. The VERIFY parameter list contains values returned from the previous call. The pointers have been updated as shown in the following figure:

```
OUTDES out1  CHARS(GT10,GB10) COPIES(1,(2,4,5))
                              |      |
                              |      PARM_PTR
                              OPER_PTR
```

Figure 77. OUTDES Statement with Adjusted Pointers Indicating Values To Be Processed

The application invokes VERIFY to validate and begin building a text unit for COPIES with its first subparameter, 1.

- Set the operand pointer field, SJVEOPEP, to OPER_PTR.
- Set the operand length field, SJVEOPEL, to 6 (length of 'COPIES').
- Set the subparameter number field, SJVEPARM, to 1.
- Set the subparameter pointer field, SJVEPRMP, PARM_PTR.
- Set the subparameter length field, SJVEPRML, to 1 (length of '1').

- Issue SJFREQ REQUEST=VERIFY,PARM=SJVEP.

VERIFY returns with a return and reason code of zero. The application updates PARM_PTR to point to the first byte of the first sublist element of the second subparameter for the COPIES keyword operand. The adjusted pointers are as follows:

```
OUTDES out1  CHARS(GT10,GB10) COPIES(1,(2,4,5))
                              |         |
                              |         PARM_PTR
                              OPER_PTR
```

Figure 78. OUTDES Statement with Adjusted Pointers for Values To Be Processed

The application invokes VERIFY to validate and continue building the COPIES text unit with the first sublist element of the second subparameter, 2.

- Set the subparameter number field, SJVEPARM, to 2.
- Set the sublist element number field, SJVESUBL, to 1.
- Set the sublist element pointer field, SJVEPRMP, to PARM_PTR.
- Set the sublist element length field, SJVEPRML, to 1 (length of '2').
- Issue SJFREQ REQUEST=VERIFY,PARM=SJVEP.

VERIFY returns with a return and reason code of zero. The text unit for the COPIES keyword operand now contains the first subparameter value, ‘1’, and the second subparameter, first sublist element value, ‘2’. The application must continue calling VERIFY with the remaining sublist elements and then specify a last call to have the COPIES keyword operand text unit completely built. The application would increment SJVESUBL to the next sublist element number and set SJVEPRMP and SJVEPRML to represent the corresponding sublist element, and call VERIFY.

Requesting last call

Once the application completes its VERIFY calls to validate and build text units for the entire OUTDES statement, it issues a separate VERIFY call to request last call. The last call must directly follow the previous example.

This example shows a VERIFY call to request the last call function. The VERIFY parameter list contains values returned from previous calls. The application does not plan to call VERIFY again.

- Set the no-cleanup bit, SJVENOCU, off. This causes the SJF environment to be freed.
- Set the last-call flag, SJVELSTC, on. This indicates to SJF to complete building the text unit for the last keyword operand or operand that was processed.

VERIFY returns with a return and reason code of zero. SJVETUPL contains the address of a text unit pointer list (in the text unit output area) that contains pointers to the text units for CHARS and COPIES.

Freeing the SJF environment (SJFREQ macro with TERMINATE)

TERMINATE frees the environment established by an SJF service. The SJF VERIFY service requires multiple invocations to complete its function and therefore must reuse the environment. An application using VERIFY may not know when the final invocation of the VERIFY service occurs (for example, when operands are processed in a loop). In this case, the application should call TERMINATE to free the environment after the final invocation. If an application has previously called

SJFREQ VERIFY with SJVENOCU set off (0) (indicating that the environment is to be freed), an application cannot call SJFREQ TERMINATE to free that environment.

Understanding the OUTDES statement

An application or its users build OUTDES statements to dynamically define processing options for a system output (SYSOUT) data set. The OUTDES statement is an alternate form of an OUTPUT JCL statement that is used in a dynamic output environment. See Chapter 27, "Dynamic output," on page 719 for more information on the dynamic output environment. The OUTDES syntax consists of the OUTDES statement, operands, keyword operands, subparameters, and sublist elements that an application or its users can request SJFREQ VERIFY to validate and build text units to represent. Each OUTDES operand and keyword operand represents an individual output processing option.

This information describes considerations for using the OUTDES statement followed by a detailed description of the syntax.

Operand and keyword operand abbreviations

Enter operands and keyword operands spelled exactly as they are shown or use an abbreviation. The minimum abbreviation for each operand and keyword operand appears following their descriptions.

For example, in the case of the BUILDING keyword operand, enter at least the first five characters. The description for the BUILDING keyword operand appears as follows:

`BUILDING('building' or building)`

specifies a building location to be used for delivery of system output. One to 60 EBCDIC text characters may be specified. See "Rules for parsing data with and without quotation marks" on page 752 for the values that may be used with and without quotation marks.

Minimum abbreviation: 5

Valid abbreviations for BUILDING are:

- BUILD
- BUILDI
- BUILDIN

Comments and line continuation

VERIFY does not account for comments or line continuation. The application is responsible for handling both comments and line continuation.

Delimiters

Blanks and commas are delimiters between elements in the syntax. Parentheses mark the bounds between a keyword operand and its subparameters and may mark the bounds of a sublist within a subparameter as well. Do not use parentheses as delimiters within sublists.

Quotation marks are delimiters for specific values within subparameters and sublists. "Rules for parsing data with and without quotation marks" on page 752 describes considerations for using quotation marks.

The application may have its own delimiters between operands; the application, however, must handle any deviations from the syntax defined in this information.

Rules for parsing data with and without quotation marks

If an application that parses an OUTDES statement encounters a keyword operand subparameter value delimited by single quotation marks, the application must do the following:

- If part of the value is two consecutive single quotation marks, convert the two single quotation marks to one single quotation mark. Two consecutive single quotation marks indicates a single quotation mark is to appear in the output. To include a single quotation mark in the string, specify two consecutive single quotation marks. For example, to obtain **Manager's Report** on the TITLE keyword, specify **TITLE('Manager''s Report')** on the OUTDES statement.
- Allow any EBCDIC text characters (X'40' to X'FE') as part of the value.

If an application that parses an OUTDES statement encounters a keyword operand subparameter value that is not delimited by single quotation marks, the application must allow any EBCDIC text characters (X'40' to X'FE') as part of the value. If the parser chooses to use any of the characters (X'40' to X'FE') as special delimiter characters, the parser would not allow these delimiter characters as part of the value. When an application specifies a keyword operand that allows quoted strings, it must specify any EBCDIC text character (X'40' to X'FE') within the string.

OUTDES statement syntax

The syntax of the OUTDES statement follows. Definitions of the operands and keyword operands follow the syntax diagram.

OUTDES

ADDRESS('address'...)
ADDRESS(address...)
ADDRESS('address1','address2',...,'address4')
ADDRESS(address1,address2,...,address4)

BUILDING('building')
BUILDING(building)

BURST **Default:** NOBURST
NOBURST

CHARS(charname,...)
CHARS
(charname1,charname2,...,charname4)

CKPTLINE(nnnnn)

CKPTPAGE(nnnnn)

CKPTSEC(nnnnn)

CLASS(output-class)

COMPACT(compaction-table-name)

CONTROL(PROGRAM)
CONTROL(SINGLE)
CONTROL(DOUBLE)
CONTROL(TRIPLE)
	Default: PROGRAM

COPIES(nnn,(group value,...)	

DATACK(BLKCHAR)
DATACK(BLKPOS)
DATACK(BLOCK)
DATACK(UNBLOCK)	

DEFAULT NODEFAULT	

DEPT('dept')
DEPT(dept)	

DEST(destination)
DEST(destination.userid)	

DPAGELBL NODPAGELBL	
FCB(fcb-name)	
FLASH(overlay,count)	
FORMDEF(member-name)	
FORMS(forms-name)	
GROUPID(output-group-name)	
INDEX(nn)	
LINDEX(nn)	
LINECT(nn)	

MODIFY(module-name)
MODIFY(module-name,trc)	

NAME('name')
NAME(name)	

NOTIFY(node.userid) NOTIFY(node.userid1, node2.userid2,...,node4.userid4)	
OUTDISP(normal-output-disp,abnormal-output-disp)	
PAGEDEF(member-name)	

PIMSG(YES,nnn)
PIMSG(NO,nnn)	

PRMODE(process-mode)	

PRTY(nnn)	

ROOM('room')
ROOM(room)	

SYSAREA NOSYSAREA	
THRESHLD(nnnnnnnn)	

TITLE('title')
TITLE(title)	

TRC NOTRC	**Default:** NOTRC
UCS(ucs-name)	
USERDATA(userdata1) USERDATA ('userdata1','userdata2',...,'userdata16')	

USERLIB(dsname)
USERLIB(dsname1, dsname2,...,dsname8)	

WRITER(external-writer-name)	

The elements of the syntax are described as follows:

ADDRESS('address'...)
ADDRESS(address...)
ADDRESS('address1','address2',...,'address4')
ADDRESS(address1,address2,...,address4)

specifies a delivery address for system output. Enter one to four address values. To skip an address value, enter a comma without specifying an address, for example, ADDRESS(,'addr2',addr4). One to 60 EBCDIC text characters may be specified for each sub-address specified. For example, if you specify ADDRESS(address1,address2), address1 and address2 may each be 60 characters. See “Rules for parsing data with and without quotation marks” on page 752 for the values that may be used with and without quotation marks.

Minimum abbreviation: 4

BUILDING('building')
BUILDING(building)

specifies a location to be used for delivery of system output. One to 60 EBCDIC text characters may be specified. See “Rules for parsing data with and without quotation marks” on page 752 for the values that may be used with and without quotation marks.

Minimum abbreviation: 5

BURST
NOBURST

BURST specifies that 3800 output is to be burst into separate sheets.

Minimum abbreviation: 3

NOBURST specifies that the printed 3800 output is to be in continuous fanfold pages. NOBURST is the default.

Minimum abbreviation: 5

CHARS(charname,...)
CHARS(charname1,charname2,...,charname4)

specifies one or more font (character arrangement) tables for printing the SYSOUT data set on a 3800 printer. You can specify up to four table names. Specify from 1 to 4 characters (alphabetic, numeric, or @, #, or $) for the character name.

Minimum abbreviation: 3

CKPTLINE(nnnnn)

specifies the maximum number of lines contained in a logical page. Specify a value from 1 through 32767. The system uses this value either for job entry subsystem (JES) checkpointing of printed output or for Systems Network Architecture (SNA) transmission checkpoints. Use CKPTLINE in combination with the CKPTPAGE operand.

If you do not specify CKPTLINE, JES2 uses an installation default specified at initialization.

Minimum abbreviation: 5

CKPTPAGE(nnnnn)

specifies the maximum number of pages to be printed or transmitted before the next SYSOUT data set checkpoint occurs. Specify a value from 1 through 32767. This value represents the number of pages to be transmitted as a single SNA chain when data is transmitted to a SNA workstation. Use CKPTPAGE in combination with the CKPTLINE operand.

If you do not specify CKPTPAGE, JES2 uses the installation default specified at initialization. The default may also indicate whether checkpoints are to be based on page count or time.

Minimum abbreviation: 5

CKPTSEC(nnnnn)

specifies the number of seconds that are to elapse between checkpoints of the SYSOUT data set that is printing. Specify a value from 1 through 32767.

If you do not specify CKPTSEC, JES2 uses the installation default specified at initialization. The default may also indicate whether checkpoints are to be based on page count or time.

Minimum abbreviation: 5

CLASS(output-class)

specifies the output class JES is to use for processing the specified SYSOUT data set. Valid output classes are characters A-Z or 0-9. The default output class is A.

Minimum abbreviation: 3

COMPACT(compaction-table-name)

specifies the name of the compaction table to be used when the data set is transmitted to a workstation. Specify a 1- to 8-character alphabetic or numeric symbolic name.

If you do not specify COMPACT, compaction is suppressed for the data set.

Minimum abbreviation: 3

CONTROL(PROGRAM)
CONTROL(SINGLE)
CONTROL(DOUBLE)
CONTROL(TRIPLE)

specifies the type of forms control to be used.

PROGRAM indicates that the carriage control character of each data record is to control line spacing on the form. PROGRAM is the default.

SINGLE indicates forced single spacing.

DOUBLE indicates forced double spacing.

TRIPLE indicates forced triple spacing.

Minimum abbreviation: 3

COPIES(nnn,(group-value,))

specifies the number of copies to be printed for the data set. The number of copies, *nnn*, can range from 1 to 255, subject to an installation limit. The default is 1.

If you use COPIES on a referenced FORMDEF operand, the system ignores the COPIES value.

If you specify group values, the system ignores the individual value, *nnn*, for the 3800 printer. The group values describe how the printed copies are to be grouped (3800 printer only). Each group value specifies the number of copies of each page that are to be grouped together. You can specify up to eight group values. For example, a group value of 3 causes the first page of a data set to be printed three times before printing is started for the second page, which will also be printed three times, and so forth.

Minimum abbreviation: 3

DATACK(BLKCHAR)
DATACK(BLKPOS)
DATACK(BLOCK)
DATACK(UNBLOCK)

specifies whether “print positioning” and “invalid character” data check errors are to be blocked or unblocked for printers accessing through the Print Services Facility (PSF).

BLKCHAR specifies character errors that are not valid are to be blocked. The errors are not reported to PSF. Print positioning errors are reported normally.

BLKPOS specifies print positioning errors are to be blocked, and not reported to PSF.

BLOCK specifies neither print positioning errors nor character errors are reported to PSF.

UNBLOCK specifies both print positioning errors and character errors are reported to PSF.

If you do not specify DATACK, the DATACK specification from the PSF PRINTDEV statement is used. If it is not specified in the PRINTDEV statement, the default is BLOCK.

Minimum abbreviation: 6

DEFAULT
NODEFAULT

DEFAULT specifies that any SYSOUT data sets that do not explicitly refer to an output descriptor (as allocated by the ALLOCATE command) should use default output characteristics when printing the data set. Minimum abbreviation: 3

NODEFAULT specifies that to use a specified set of output characteristics, an explicit reference to the output descriptor needs to be made on the ALLOCATE command when allocating a SYSOUT data set.

Minimum abbreviation: 5

DEPT('dept')
DEPT(dept)

specifies a department identifier that is to be associated with system output. The identifier is used to aid in the distribution of output. One to 60 EBCDIC text characters may be specified. See “Rules for parsing data with and without quotation marks” on page 752 for the values that may be used with and without quotation marks.

Minimum abbreviation: 4

DEST(destination)
DEST(destination.userid)

specifies a remote workstation or a user at a specific remote workstation to which the output is routed for processing. You can specify from 1 to 8 characters for either *destination* or *userid*.

Minimum abbreviation: 3

DPAGELBL
NODPAGELBL

DPAGELBL specifies that the system is to print a security-related character string on each page of output. The character string is associated with a security label (usually the security label of the user's current session). Your installation determines the character string used.

Minimum abbreviation: 6

NODPAGELBL specifies that the system-generated security character string is to be suppressed on the pages of output. You must have the appropriate RACF access authority to override page labeling. If you need to override DPAGELBL but are unable to, check your installation security procedures or see your RACF security administrator.

Minimum abbreviation: 8

FCB(fcb-name)

specifies the name of the forms control buffer (FCB) or image to be used for the 3211, 3203-5, or 3800 printers. The name of the FCB is a 1- to 4-alphabetic or numeric string consisting of the last 1 to 4 characters of the following:

- FCB2xxxx member for the 3211 or 3203-5 printer or printers supported by System Network Architecture (SNA)
- FCB3xxxx member for the 3800 printer.

Minimum abbreviation: 3

FLASH(overlay,count)

specifies the name of the forms overlay to be used on the 3800. The overlay is “flashed” on a form or other printed information over each page of output. The overlay name must be 1 to 4 characters (alphabetic, numeric, or @, #, or $).

FLASH also allows you to specify the number of copies (count) on which the overlay is to be printed. The count can range from 0 to 255. To flash **no** copies, specify a count of zero.

Minimum abbreviation: 3

FORMDEF(member-name)

specifies the member name of a partitioned data set containing information that the Advanced Function Printer (AFP 3800-3 or 3800-8) uses to print a data set. The member can contain the following information:

- The overlays that are to be invoked during output processing
- The location on the page where the overlays are to be placed
- The suppressions that can be activated for specified page formats.

The member name contains a maximum of 6 characters, of which the first two are predefined by your installation. For the last four characters, specify alphabetic or numeric characters, or the characters @, #, or $.

Minimum abbreviation: 5

FORMS(forms-name)

specifies the name of the form on which the output is to be printed. Specify 1 to 8 characters (alphabetic, numeric or @, #, or $) for the forms name.

If you do not specify FORMS, JES uses the installation default specified at initialization.

Minimum abbreviation: 5

GROUPID(output-group-name)

specifies the name to be used by JES2 to identify which of a job's SYSOUT data sets are to form an output group. The output group name consists of 1 to 8 alphabetic or numeric characters and is selected by the system programmer to define an output group for the job.

Minimum abbreviation: 3

INDEX(nn)

specifies a value indicating the data set indexing print offset (to the right) for the 3211 printer with the indexing feature. The width of the print line is reduced by the value of INDEX. Specify a value from 1 through 31. The value 1 indicates flush left. The values 2 through 31 indent the print line by nn-1 positions.

The default is 1, which indicates flush left.

Minimum abbreviation: 3

LINDEX(nn)

specifies a value indicating the data set indexing print offset (to the left) for the 3211 printer with the indexing feature. The width of the print line is reduced by the value of LINDEX. Specify a value from 1 through 31. The value 1 indicates flush right. The values 2 through 31 move the right margin over by nn-1 positions.

The default is 1, which indicates flush right. LINDEX is ignored on printers other than the 3211 printer.

Minimum abbreviation: 4

LINECT(nnn)

specifies the number of lines that are to be printed before overflow processing. Specify a value from 0 through 255. If you specify zero, no overflow processing is done.

If you do not specify LINECT, JES2 obtains the value from one of the following:

- The linect field of the accounting information parameter on the JCL JOB statement.
- The installation default specified at JES2 initialization.

Minimum abbreviation: 4

MODIFY(module-name,trc)

specifies the name of a copy modification module, which is loaded into the 3800 printing subsystem. This module contains predefined data such as legends, column headings, or blanks. The module specifies where and on which copies the data is to be printed. The module is in the SYS1.IMAGELIB system data set. Specify 1 to 4 characters (alphabetic, numeric, or @, #, or $) for the module name.

The table reference character (TRC), which is optional, corresponds to the character set or sets specified on the CHARS operand. Values are 1 to 4 characters (alphabetic, numeric, or @, #, or $).

Minimum abbreviation: 3

NAME('name')
NAME(name)

specifies a name that is to be associated with system output. The name is used by your installation to help in the distribution of output. One to 60 EBCDIC text characters may be specified. See "Rules for parsing data with and without quotation marks" on page 752 for the values that may be used with and without quotation marks.

Minimum abbreviation: 4

NOTIFY(node.userid)
NOTIFY (node1.userid1, node2.userid2,...,node4.userid4)

specifies the node and userid for the recipients of the print complete messages that are issued when a data set completes printing. The message indicates whether the job completed successfully and identifies the output that has completed printing. Node may be omitted from any destination. If node is not specified, it is assumed to be the node where the job was submitted. You may specify up to four recipients of print complete messages.

If you do not specify NOTIFY, no print complete message is issued.

Minimum abbreviation: 3

OUTDISP(normal-output-disp,abnormal-output-disp)

specifies the disposition of a SYSOUT data set. Possible dispositions include the various combinations of holding the data set before or after writing it to a printer or an external writer. Normal-output-disp is the disposition for the data set if the job completes normally. Abnormal-output-disp is the disposition for the data set if the job completes abnormally. Both normal-output-disp and abnormal-output-disp are optional, but one should be specified.

Minimum abbreviation: 5

The following table lists the valid values for normal-output-disp and abnormal-output-disp.

Table 98. Values for Normal- and Abnormal-Output-Disp

Value	Description
WRITE	System output is to be processed and deleted.
HOLD	System output is to be held until released by the user or operator. When the system output is released, its disposition is changed to WRITE.
KEEP	System output is to be processed, and upon completion of processing, its disposition is to be changed to LEAVE
LEAVE	System output is to be held until it is released by the user or operator. When the system output is released, its disposition is changed to KEEP.
PURGE	System output is deleted without processing.

PAGEDEF(member-name)

specifies the member of a partitioned data set containing information that the Advanced Function Printer uses to print the data set. The member can contain the following information:

- Logical page size and width
- Fonts
- Page segments
- Multiple page types or formats
- Lines within a page; for example, line origin, carriage controls, and spacing
- Multiple logical pages on a physical page.

The member name contains a maximum of 6 characters, of which the first two are predefined by your installation. For the last four characters, specify alphabetic or numeric characters, or the characters @, #, or $.

Minimum abbreviation: 3

PIMSG(YES,nnn)
PIMSG(NO,nnn)

specifies whether messages are to be printed. Values are 0 through 999. The value specifies that the system is to cancel the printing of the current data set after the specified number of errors have been either:

- Detected by the functional subsystem (FSS)
- Reported to FSS by the printer.

PIMSG(YES) specifies that messages generated by FSS are to be printed. PIMSG(YES,16) is the default.

PIMSG(NO) specifies that messages are to be suppressed.

If you specify nnn as zero, the system does not cancel the printing of the current data set.

Minimum abbreviation: 3

PRMODE(process-mode)

specifies the process mode to be used to schedule output data sets either to output devices running under a functional subsystem (FSS) or to an output device managed by JES. For a list of valid process modes, contact your system programmer. If you do not specify PRMODE, JES might determine the process mode based upon the content of the data. Specify 1 to 8 alphabetic or numeric characters for the process mode.

Use PRMODE to indicate the type of processing you want for a data set. You can use it to direct JES scheduling of this data set to a particular output FSS or

JES writer. You can also use PRMODE to request specific processing of a Network Job Entry (NJE) transmitted data set at the destination node without knowing the device name or a SYSOUT class.

Minimum abbreviation: 3

PRTY(nnn)

specifies the initial selection priority for the data set. Specify a value from 0 through 255, where 0 is the lowest output processing priority and 255 is the highest output processing priority.

Minimum abbreviation: 4

ROOM('room')
ROOM(room)

specifies a room identifier that is to be associated with system output. The room is used by your installation to help in the distribution of output. One to 60 EBCDIC text characters may be specified. See "Rules for parsing data with and without quotation marks" on page 752 for the values that may be used with and without quotation marks.

Minimum abbreviation: 4

SYSAREA
NOSYSAREA

SYSAREA specifies that the system printable area is to be reserved for printing a character string associated with a security label. SYSAREA is the default.

Minimum abbreviation: 7

NOSYSAREA specifies that you want to print on the entire page of output (including the system printable area) for the current printing. In this case, the system area is not reserved for printing a security character string. You must have the appropriate RACF access authority to override page labeling. If you need to override the system area but are unable to, check your installation security procedures or see your RACF security administrator.

Minimum abbreviation: 9

THRESHLD(nnnnnnnn)

specifies a maximum size print work unit or lines for SYSOUT data sets on a job level, step level, or SYSOUT data set level. Specify a value between 1 to 99999999. When this size is exceeded, a new unit of work is created for print scheduling.

Units of work are established on data set boundaries. These boundaries allow multiple printers to print different data sets of a job at the same time. If the number specified on THRESHLD is exceeded, any excess data sets are queued as a separate unit of work. THRESHLD assumes that the data set size is the number of records in the data set multiplied by the number of copies.

If you do not specify THRESHLD, JES3 uses the installation default specified at initialization.

Minimum abbreviation: 3

TITLE('title')
TITLE(title)

specifies a report title or description that is to be included on the separator pages of system output. The title is used by your installation to help in the distribution of output. One to 60 EBCDIC text characters may be specified. See "Rules for parsing data with and without quotation marks" on page 752 for the values that may be used with and without quotation marks.

Minimum abbreviation: 5

TRC
NOTRC

TRC specifies whether the data records contain table reference character (TRC) codes. The codes identify the font to be used to print each record.

A TRC code immediately follows the carriage control character, if any. Its value corresponds to either one of the four fonts specified by CHARS or one of the fonts in the PAGEDEF font list. PAGEDEF allows more than four fonts to be specified.

Minimum abbreviation: 3

NOTRC specifies that the data set does not contain TRC codes. NOTRC is the default.

Minimum abbreviation: 5

UCS(ucs-name)

specifies the name for the universal character set. Specify up to 4 alphabetic or numeric characters, or the characters @, #, or $. If you process the print data set through PSF and do not specify CHARS, the system uses the UCS as the font name.

Minimum abbreviation: 3

USERDATA('userdata1','userdata2',...,'userdata16')
USERDATA(userdata1)

specifies the installation-defined values for the installation's prescribed processing. Refer to your installation's definition on the intent and use of this keyword operand.

```
USERDATA(value,...)
```

You can code up to 16 installation-defined values. Each value may be from 1 to 60 EBCDIC text characters. Null positions in the USERDATA parameter are not allowed. For example, you cannot code USERDATA(,value) or USERDATA(value,,value). Each value may be enclosed in apostrophes; however, apostrophes around each value are not required.

- Valid characters in enclosing apostrophes:
  - A USERDATA parameter value enclosed in apostrophes can contain any EBCDIC text character.
  - Enclose a value that contains a blank in apostrophes.
  - To code an apostrophe as part of the parameter value, code 2 apostrophes, and enclose the entire value in single apostrophes. For example:

    ```
    OUTDES name NEW USERDATA ('USERKEY1=User''s value')
    ```
- Valid characters without enclosing apostrophes:
  - It can contain any character other than a blank, comma, tab, or semicolon.

USERLIB(dsname)
USERLIB(dsname1, dsname2,...,dsname8)

specifies the data set name of a library containing AFP resources. The data set must be a fully-qualified cataloged data set. The resources in the library specify how the SYSOUT data set is to be printed and include the following:

- Fonts
- Page segments
- Overlays

- Pagedefs
- Formdefs

You may specify up to eight libraries.

A valid data set name is 1 to 44 characters (alphabetic, numeric, or @, #, $). You may specify up to 22 levels separated by periods. The first character of each level must be alphabetic or @, #, or $.

Each data set containing a library is concatenated to the system resource libraries. These data sets specified with USERLIB are checked first for requested resources.

If you do not specify USERLIB, the system and installation print resources are used.

Minimum abbreviation: 3

WRITER(external-writer-name)
specifies the member name of an installation-written program in the system library that is to write the SYSOUT data set, instead of JES2 or JES3. If you specify the external writer name, the output data set is written under the control of that external writer rather than the control of JES2 or JES3. The writer name can contain 1 to 8 characters (alphabetic, numeric or @, #, or $).

Minimum abbreviation: 3

Chapter 29. Processing user trace entries in the system trace table

References

For a description of the PTRACE macro, see *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*.

Formatting a USRn trace table entry

The following brief explanation describes the process involved in formatting a USRn trace table entry (TTE) using the IBM-supplied routines.

The system trace filter/formatter module, IEAVETEF, uses the home ASID in the TTE to determine if a particular trace table entry is selected for formatting. If so, the system formats the following system-supplied status information for the TTE in the trace output buffer.

Note: The labels in parentheses are those mapped by macro IHATROB.

- The processor number (TROBPRID)
- The HASID (TROBASID)
- The current TCB address (TROBTCBA)
- The USRn acronym for the TTE (TROBID)
- The return address of the issuer of PTRACE (TROBRET)
- The continuation information, if the TTE is part of a multi-part entry (TROBUNQ1, TROBUNQ2, and TROBUNQ3)
- The PASID (TROBPASN)
- The SASID (TROBSASN)
- The time-of-day value for the TTE (TROBTIME)

The system then calls the appropriate USRn formatting routine (ITRF0n7F), which calls ITRFDEFU, the default formatting routine.

ITRFDEFU formats the user data, in hexadecimal, in the trace output buffer fields labelled TROBUNQ1, TROBUNQ2, and TROBUNQ3. These fields correspond to the columns headed by UNIQUE-1, UNIQUE-2, and UNIQUE-3 in the printed trace table. The system then calls the print buffer service routine to print the output line. A single USRn TTE can contain up to five fullwords of user data, which are formatted on two successive lines in the printed output.

Replacing a USRn TTE formatting routine

You may replace any of the ITRF0n7F formatting routines with one that fits your installation's requirements, link editing it into load module IEAVETFC in SYS1.LPALIB. Generally, your routine must conform to the same conventions and requirements that the ITRF0n7F routines supplied by IBM follow. This information describes some of those conventions.

Parameters passed to the USRn formatter

When the system passes control to a USRn formatting routine, the output registers contain the following values:

Register	Contents
0	Used as a work register by the system
1	Address of a parameter list
2-12	Used as work registers by the system
13	Address of save area
14	Return address
15	Address of entry point

The parameter list addressed by register 1 contains:

1. Address of a **token**. The print buffer service routine passes the token each time the formatting routine calls it.
2. Address of the **trace output buffer**. IHATROB maps the trace output buffer which is initialized with the status information and, if the TTE is a multi-part entry, with continuation information in TROBUNQ1, which corresponds to the UNIQUE-1 column of the formatted output line.
3. Address of the **USRn TTE**.
4. Address of a **512-byte work area**. The USRn formatting routines must be reentrant. The work area received as the fourth parameter allows this. Your formatting routine may use the 512 bytes for any purpose; IBM recommends that it be used as an automatic data area. The same work area is passed to each USRn formatting routine; it is not cleared between calls. It is, however, initialized to zeroes before the first call to a USRn routine.
5. Address of a byte containing the **subpool number** to be used for additional work space. If your USRn routine needs more than 512 bytes for its processing, it can obtain more storage through the GETMAIN macro. The fifth parameter is the subpool that must be used for this GETMAIN request.

Note: If your routine does issue a GETMAIN, be sure that it also issues a FREEMAIN for that storage. If it does not free the storage reserved for its use when it is running on behalf of a SNAP dump request, the storage it reserves will remain allocated and unavailable for the life of the job.

Return codes from the USRn formatter

When your USRn formatting routine returns control to the system, it must put a return code into register 15, based on whether or not it formatted the TTE. The possible codes and their meanings are:

Code	Meaning
0	The USRn TTE was formatted
4	The USRn TTE was not formatted

The user should restore registers 0 through 13.

Printing the trace output buffer contents

Your routine must update the trace output buffer with the user data and call the print buffer service routine, IEAVETPB, to print each output line.

IEAVETPB, entry point in module IEAVETFA, prints the trace output buffer. In the case of a print dump request, IEAVETPB also keeps track of the number of lines printed on a page and skips to a new page when the maximum has been printed or the TTE being printed requires more than the lines left on the page.

When your USRn formatting routine passes control to IEAVETPB, the input registers must contain the following values:

Register	**Contents**
1	Address of a five-fullword parameter list
13	Address of save area
14	Return address
15	Address of entry point

The parameter list in register 1 contains:

1. Address of the token received from the system. If it does not receive the token, it issues a X'09E' ABEND with a reason code of X'00005301', and the system trace formatter terminates.
2. Address of the trace output buffer (TROB). It locates the output line to be printed using the address in the second parameter.
3. Address of the relative output line number. The third parameter is the relative output line number for the formatting of a single TTE. The value in this parameter indicates which line this is of the total number of output lines needed to format the TTE.
4. Address of the number of output lines expected. The fourth parameter is the total number of lines needed to format the TTE; that is, the number of times the USRn formatting routine will call IEAVETPB to print a particular TTE.
5. Address of the print option for this call. The fifth parameter indicates the kind of output contained in the trace output buffer:
   - X'80000000' means the output buffer contains a TTE
   - X'00000000' means the output buffer contains a message

Having printed a line of output, IEAVETPB returns to the USRn formatting routine with a return code of zero in register 15.

Handling errors during TTE formatting

If your USRn formatting routine encounters a program check, the ESTAE for the system trace formatter controller gets control. The ESTAE tests the completion code. If it is X'0C6' through X'0CF', indicating a likely data-dependent arithmetic or conversion error, the following takes place:

- A message is printed in the trace table output saying that the USRn format routine failed and is disabled.
- The USRn TTE that caused the failure is formatted by ITRFDEFU, the default USRn formatter routine.
- Future USRn TTEs that would have been formatted by the failing routine will also be formatted.

If the completion code is other than X'0C6' through X'0CF', IEAVETFC terminates and a message is printed saying that the trace formatter failed because of an unrecoverable error.

Figure 79 is a sample of the assembler language code needed to format a USRn trace table entry. The sample CSECT formats a USR0 TTE that was created by the following PTRACE:

```
PTRACE TYPE=USR0,REGS=(2,4),SAVEAREA=STANDARD
```

```
ITRF007F  CSECT ,
ITRF007F  AMODE 31
ITRF007F  RMODE ANY
*---------------------------------REGISTER EQUATES
R0        EQU   0
R1        EQU   1
R2        EQU   2
TTEPTR    EQU   7                 TTE ADDRESSABILITY
R8        EQU   8                 MODULE ADDRESSABILITY
R9        EQU   9                 DATA AREA ADDRESSABILITY
TROBPTR   EQU   12                TROB ADDRESSABILITY
R12       EQU   12
R13       EQU   13
R14       EQU   14
R15       EQU   15
*---------------------------------STANDARD ENTRY LINKAGE
          DS    0H
          USING *,R15             TEMPORARY MODULE ADDRESSABILITY
          DROP  R15               DROP TEMPORARY ADDRESSABILITY
          STM   R14,R12,12(R13)   SAVE REGISTERS
          BALR  R8,0
          USING *,R8              MODULE ADDRESSABILITY
******************************************************************************
*         OBTAIN ADDRESSABILITY TO THE PARAMETERS                            *
******************************************************************************
          L     R9,12(,R1)        ADDRESS OF THE AUTOMATIC DATA AREA
*                                 IS THE FOURTH PARAMETER
          USING DATA,R9           DATA AREA ADDRESSABILITY
          ST    R13,SAVE0001+4    BACKWARD CHAIN SAVEAREAS
          LA    R2,SAVE001        ADDRESS OF MODULE SAVEAREA
          ST    R2,8(R13)         FORWARD CHAIN SAVEAREAS
          LR    R13,R2            POINT TO CURRENT SAVEAREA
*---------------------------------SAVE THE TOKEN
          L     R2,0(,R1)         ADDRESS OF THE TOKEN
          ST    R2,TOKEN          SAVE TOKEN IN AUTOMATIC AREA
*---------------------------------TRACE OUTPUT BUFFER ADDRESSABILITY
          L     TROBPTR,4(,R1)    GET TROB ADDRESS FROM PARMLIST
          USING TROB,TROBPTR      TROB ADDRESSABILITY
*---------------------------------USR0 TTE ADDRESSABILITY
          L     TTEPTR,8(,R1)     ADDRESS OF THE CURRENT TTE
          USING TTE,TTEPTR        SET USR0 TTE ADDRESSABILITY
```

Figure 79. Sample Code for Formatting USRn Trace Table Entries

```
************************************************************************
*         FORMAT A HIGHLIGHTING MESSAGE                                *
************************************************************************
          L     R2,TOKEN          TOKEN TO BE PASSED
          ST    R2,ETPBLIST       SET 1ST PARAMETER(TOKEN)
          LA    R2,HILITE         120 CHAR MESSAGE TO BE OUTPUT
          ST    R2,ETPBLIST+4     SET 2ND PARAMETER
          LA    R2,LINE1          RELATIVE LINE NUMBER OF THE LINE
*                                 TO BE PRINTED
          ST    R2,ETPBLIST+8     SET 3RD PARAMETER
          LA    R2,MAXLINES       NUMBER OF LINES OF OUTPUT EXPECTED
          ST    R2,ETPBLIST+12    SET 4TH PARAMETER
          LA    R2,CPMSG          THE OUTPUT IS A MESSAGE
          ST    R2,ETPBLIST+16    SET 5TH PARAMETER
          LA    R1,ETPBLIST       LOAD ADDRESS OF PARAMETER LIST
          L     R15,IEAVETPB      LOAD ADDRESS OF IEAVETPB ROUTINE
          BALR  R14,R15           CALL IEAVETPB
***********************************************************************
*         INITIALIZE THE OUTPUT BUFFER WITH USR0 DATA WORDS           *
***********************************************************************
          MVC   WORK5,TTEWRD5     MOVE USER WORD TO WORK AREA
          UNPK  WORK10,WORK5      UNPACK USER DATA WORD 1
          TR    WORK10,EBCTABL    TRANSLATE TO PRINTABLE HEX
          MVC   TROBUNQ1,WORK10+1 MOVE TO OUTPUT BUFFER UNIQUE1 COLUMN
          MVC   WORK5,TTEWRD6     MOVE USER WORD TO WORK AREA
          UNPK  WORK10,WORK5      UNPACK USER DATA WORD 2
          TR    WORK10,EBCTABL    TRANSLATE TO PRINTABLE HEX
          MVC   TROBUNQ2,WORK10+1 MOVE TO OUTPUT BUFFER UNIQUE2 COLUMN
          MVC   WORK5,TTEWRD7     MOVE USER WORD TO WORK AREA
          UNPK  WORK10,WORK5      UNPACK USER DATA WORD 3
          TR    WORK10,EBCTABL    TRANSLATE TO PRINTABLE HEX
          MVC   TROBUNQ3,WORK10+1 MOVE TO OUTPUT BUFFER UNIQUE3 COLUMN
***********************************************************************
*         FORMAT THE USR0 TRACE TABLE ENTRY                           *
***********************************************************************
          L     R2,TOKEN          TOKEN TO BE PASSED
          ST    R2,ETPBLIST       SET 1ST PARAMETER(TOKEN)
          LA    R2,TROB           TROB TO BE OUTPUT
          ST    R2,ETPBLIST+4     SET 2ND PARAMETER
          LA    R2,LINE1          RELATIVE LINE NUMBER OF THE LINE
*                                 TO BE PRINTED
          ST    R2,ETPBLIST+8     SET 3RD PARAMETER
          LA    R2,MAXLINES       NUMBER OF LINES OF OUTPUT EXPECTED
          ST    R2,ETPBLIST+12    SET 4TH PARAMETER
          LA    R2,CPTTE          THE OUTPUT IS A PART OF A TTE
          ST    R2,ETPBLIST+16    SET 5TH PARAMETER
          LA    R1,ETPBLIST       LOAD ADDRESS OF PARAMETER LIST
          L     R15,IEAVETPB      LOAD ADDRESS OF IEAVETPB ROUTINE
          BALR  R14,R15           CALL IEAVETPB

***********************************************************************
*        RETURN TO THE CALLER WITH A RETURN CODE OF 0                 *
***********************************************************************
         EXIT  LA R15,0           LOAD UP THE RETURN CODE
         L     R13,4(R13)         LOCATE CALLERS SAVE AREA
         L     R14,12(R13)        RESTORE THE RETURN ADDRESS
         LM    R0,R12,20(R13)     RESTORE REGISTERS
         BR    14
*---------------------------------CONSTANTS
         DS    0D
EBCTABL  EQU   *                  TRANSLATE TABLE FOR PRINTABLE HEX
         ORG   *+240
         DC    C'0123456789ABCDEF'
LINE1    DC    F'1'               RELATIVE LINE NUMBER 1 FOR HILITE
MAXLINES DC    F'1'               TOTAL NUMBER OF OUTPUT LINES
HILITE   DC    CL120'*****************THE MUCH AWAITED USR0 TRACE EVENT
               HAS OCCURRED! ********************************'
*---------------------------------
CPTTE    DC    X'80000000'        IEAVETPB OPTIONS WORD VALUE
*                                 THE OUTPUT IS A TTE
CPMSG    DC    X'01000000'        IEAVETPB OPTIONS WORD VALUE
*                                 THE OUTPUT IS A MESSAGE
*---------------------------------
IEAVETPB DC    V(IEAVETPB)        ADDRESS OF IEAVETPB ROUTINE
*---------------------------------DYNAMIC DATA AREA
DATA     DSECT
SAVE0001 DS    18F                STANDARD SAVEAREA
ETPBLIST DS    5F                 IEAVETPB PARAMETER LIST
```

```
TOKEN    DS    F                 ADDRESS OF THE TOKEN TO BE PASSED
*                                TO IEAVETPB
WORK5    DS   CL5                INPUT WORK AREA FOR USER DATA
         DS   0F
WORK10   DS   CL10               OUTPUT WORK AREA FOR USER DATA
ENDDATA  EQU  *
ITRF007F CSECT
SIZDATA  DC   AL4(((ENDDATA-DATA+7)/8)*8)
         IHATROB
         IHATTE USRN=YES
         END
```

Chapter 30. Using system logger services

When an application is authorized (supervisor state, system PKM) you can use system logger services that can be used in addition to the general system logger services available to both authorized and unauthorized programs. See *z/OS MVS Programming: Assembler Services Guide* for the general description of system logger and the services that can be used by any program.

This topic covers the information you need to write a system logger application, as follows:

- "What is system logger?"
- "The system logger configuration" on page 775
- "Overview of authorized system logger services" on page 777.
- "IXGCONN: Connecting to and disconnecting from a log stream" on page 782
- "IXGDELET: Deleting log blocks from a log stream" on page 788.
- "Setting up the system logger configuration" on page 788.
- "When things go wrong: Recovery scenarios for system logger" on page 797.

What is system logger?

System logger is a set of services that allows an application to write, browse, and delete log data. You can use system logger services to merge data from multiple instances of an application, including merging data from different systems across a sysplex.

For example, suppose you are concurrently running multiple instances of an application in a sysplex, and each application instance can update a common database. It is important for your installation to maintain a common log of all updates to the database from across the sysplex, so that if the database should be damaged, it can be restored from the backup copy. You can merge the log data from applications across the sysplex into a **log stream**, which is simply a collection of data in **log blocks** residing in the coupling facility and on DASD (see Figure 80 on page 772).

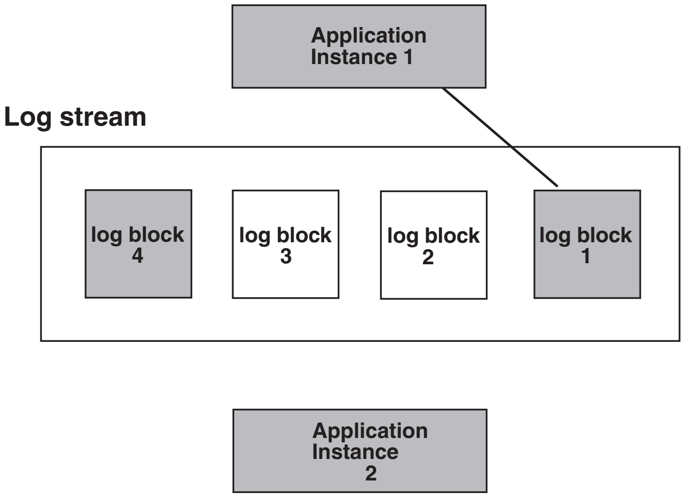


Figure 80. System Logger Log Stream

The log stream

A log stream is an application specific collection of data that is used as a log. The data is written to and read from the log stream by one or more instances of the application associated with the log stream. A log stream can be used for such purposes as a transaction log, a log for re-creating databases, a recovery log, or other logs needed by applications.

A system logger application can write log data into a **log stream**, which is simply a collection of data. Data in a log stream spans two kinds of storage:

- **Interim storage**, where data can be accessed quickly without incurring DASD I/O.
- **DASD log data set storage**, where data is hardened for longer term access. When the interim storage medium for a log stream reaches a user-defined threshold, the log data is offloaded to DASD log data sets.

There are two types of log streams:

- Coupling facility log streams
- DASD-only log streams.

The main difference between the two types of log streams is the storage medium system logger uses to hold interim log data:

- In a coupling facility log stream, interim storage for log data is in coupling facility list structures. See “Coupling facility log stream” on page 773.
- In a DASD-only log stream interim storage for log data is contained in local storage buffers on the system. **Local storage buffers** are data space areas associated with the system logger address space, IXGLOGR. See “DASD-only log stream” on page 774.

Your installation can use just coupling facility log streams, just DASD-only log streams, or a combination of both types of log streams. The requirements and preparation steps for the two types of log streams are somewhat different; see “Setting up the system logger configuration” on page 788.

Some key considerations for choosing either coupling facility log steams or DASD-only log streams are:

- The location and concurrent activity of writers and readers to a log stream's log data
- The volume of log data written to a log stream.

Coupling facility log streams are required when:

1. There needs to be more than one concurrent log writer and/or log reader to the log stream from more than one system in the sysplex.
2. There are high volumes of log data being written to the log stream.

DASD-only log streams can be used when:

1. There is not a need to have more than one concurrent log writer and/or log reader to the log stream from more than one system in the sysplex.
2. There are low volumes of log data being written to the log stream.

Note: Since DASD-only log streams always use staging data sets, high volume writers of log data may be throttled back by the I/O required to record each record sequentially to the log stream's staging data sets.

With z/OS Release 3 and higher, you can also upgrade existing structure-based log streams to use a different coupling facility structure. See the topic "Upgrading an Existing Structure-based log stream" in *z/OS MVS Programming: Assembler Services Guide*.

Coupling facility log stream

Figure 81 shows how a coupling facility log stream spans two levels of storage; the coupling facility for interim storage and DASD log data sets for more permanent storage. When the coupling facility space for the log stream fills, the data is offloaded to DASD log data sets. A coupling facility log stream can contain data from multiple systems, allowing a system logger application to merge data from systems across the sysplex.

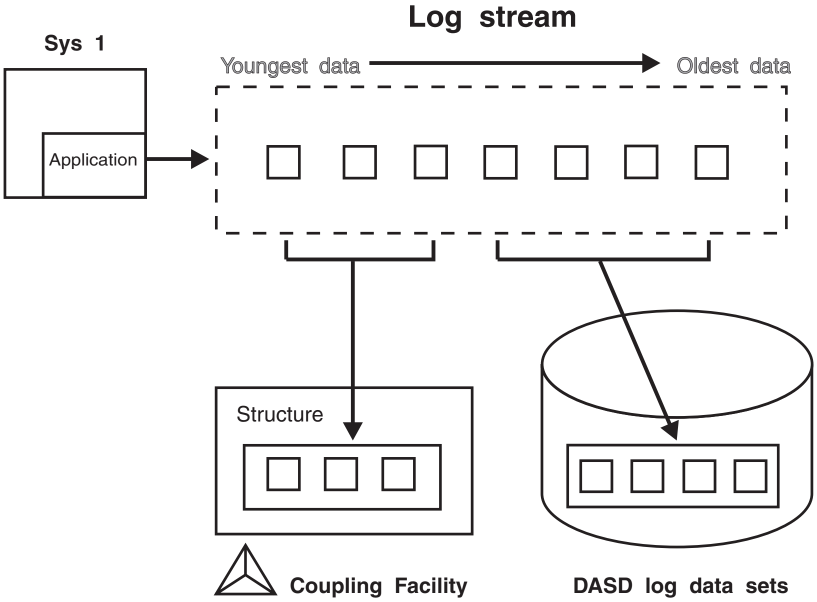


Figure 81. Log Stream Data on the Coupling Facility and DASD

When a system logger application writes a log block to a coupling facility log stream, system logger writes it first to a coupling facility list structure. System logger requires that a coupling facility list structure be associated with each log stream. When the coupling facility structure space allocated for the log stream reaches the installation-defined threshold, system logger moves (**offloads**) the log blocks from the coupling facility structure to VSAM linear DASD data sets, so that the coupling facility space for the log stream can be used to hold new log blocks. From a user's point of view, the actual location of the log data in the log stream is transparent.

DASD-only log stream

Figure 82 shows a DASD-only log stream spanning two levels of storage; local storage buffers for interim storage, which is then offloaded to DASD log data sets for more permanent storage.

A DASD-only log stream has a single-system scope; only one system at a time can connect toa DASD-only log stream. Multiple applications from the same system can, however, simultaneously connect to a DASD-only log stream.

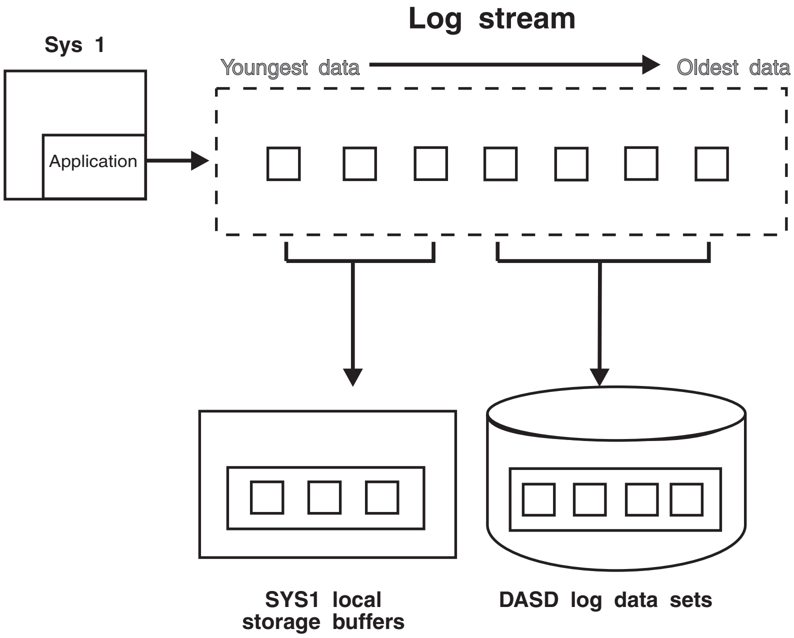


Figure 82. Log Stream Data in Local Storage Buffers and DASD Log Data Sets

When a system logger application writes a log block to a DASD-only log stream, system logger writes it first to the local storage buffers for the system and duplexes it to a DASD staging data set associated with the log stream. When the staging data set space allocated for the log stream reaches the installation-defined threshold, system logger offloads the log blocks from local storage buffers to VSAM linear DASD data sets. From a user's point of view, the actual location of the log data in the log stream is transparent.

Both a DASD-only log stream and a coupling facility log stream can have data in multiple DASD log data sets; as a log stream fills log data sets on DASD, system logger automatically allocates new ones for the log stream.

The system logger configuration

See *z/OS MVS Setting Up a Sysplex* for a complete description of the System Logger Configuration and setup requirements.

The system logger configuration you use depends on whether or not you use a coupling facility.

Coupling facility log stream configuration: Figure 83 shows all the parts involved when a system logger application writes to a coupling facility log stream. In this example, a system logger application runs on two systems in a sysplex. Both instances of the application write data to the same log stream, TRANSLOG. Each system contains a system logger address space. A system logger application uses system logger services to access the system logger capabilities.

When a system logger application writes data to a coupling facility log stream, system logger writes the data to a coupling facility list structure associated with the log stream. Then, when the coupling facility structure fills with data, system logger offloads the data to DASD log data sets.

You can optionally elect to have coupling facility data duplexed to DASD staging data sets for a coupling facility log stream.

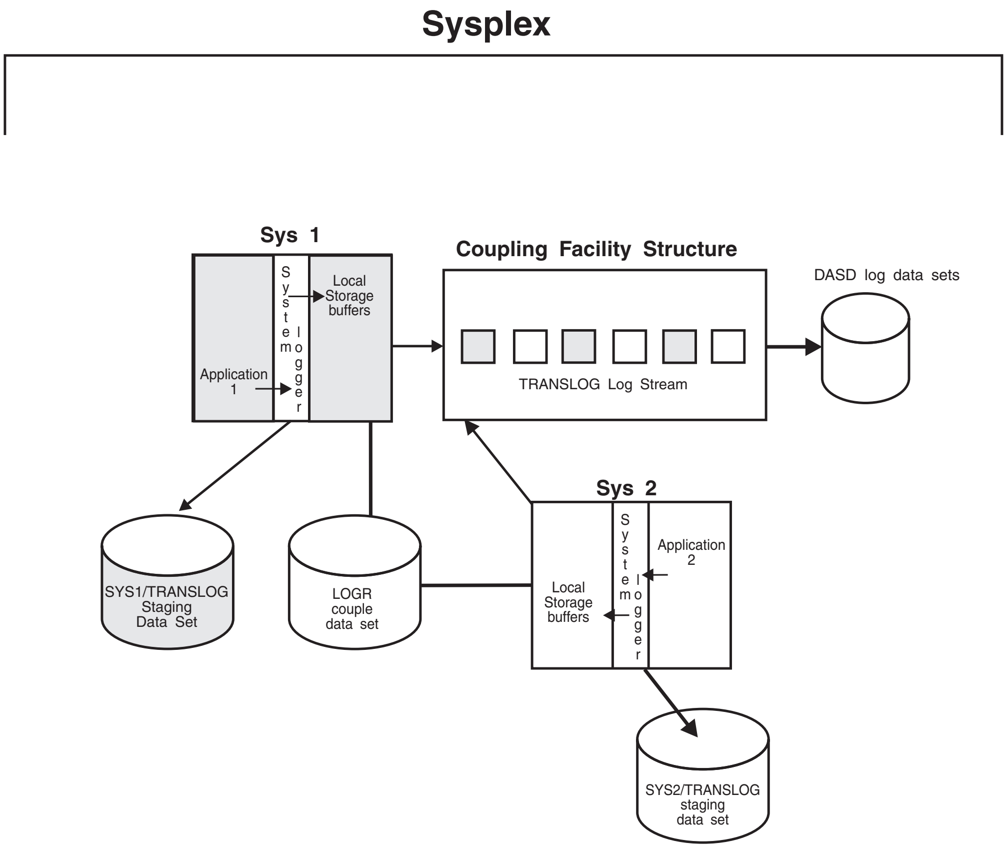


Figure 83. A Complete Coupling Facility Log Stream Configuration

DASD-only log stream configuration: Figure 84 on page 776 shows all the parts involved when a system logger application writes to a DASD-only log stream. System logger writes the data to the local storage buffers on the system, duplexing it at the same time to the DASD staging data sets associated with the log stream.

Then, when the staging data set fills with data, system logger offloads the data to DASD log data sets. Duplexing to DASD staging data sets is an option for a coupling facility log stream; it is a required automatic part of a DASD-only log stream. A system logger application uses system logger services to access the system logger capabilities.

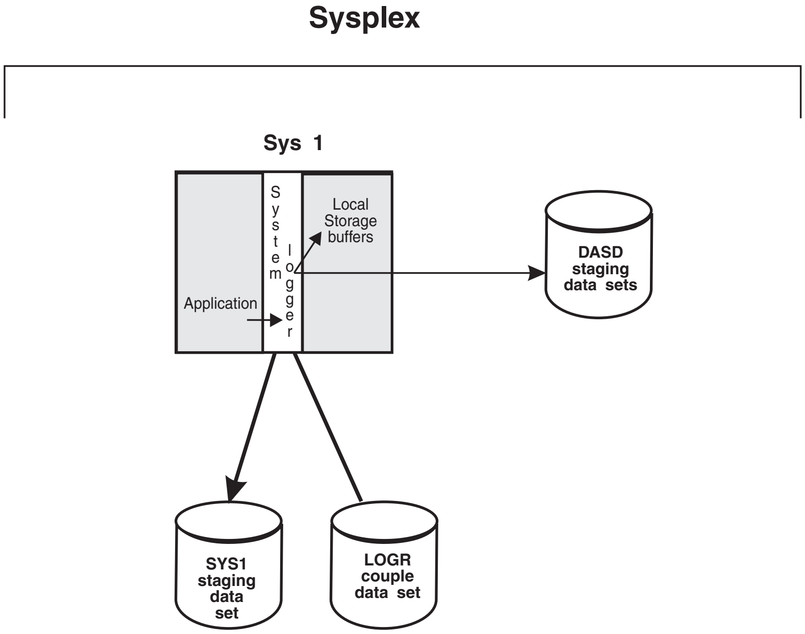


Figure 84. A DASD-Only Configuration

The system logger component

The system logger component resides in its own address space on each system in a sysplex. Some of the component processing will differ, depending on whether a given log stream is a coupling facility log stream or a DASD-only log stream. The system logger component does the following:

- Provides a set of system services that allows a system logger application to use the system logger component. See *z/OS MVS Programming: Assembler Services Reference IAR-XCT* and the *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.
- Maintains information in the LOGR policy about the current use of log streams and if used, coupling facility list structures.
- For coupling facility log streams, system logger interacts with cross-system extended services (XES) to connect to and use the coupling facility for system logger applications.
- Obtains local storage buffer space. For a coupling facility log stream, local storage buffers can be used for duplexing log data. For a DASD-only log stream, local storage buffers are used as interim storage for log data before it is offloaded to DASD log data sets.
- Offloads data to DASD log data sets as follows:

  For coupling facility log streams, system logger offloads log data from the coupling facility to DASD log data sets as the coupling facility structure space associated with the log stream reaches the installation-defined thresholds.

  For DASD-only log streams, system logger offloads log data from the local storage buffers to DASD log data sets as the DASD staging data set space reaches the installation-defined thresholds.

- Automatically allocates new DASD log data sets for log streams.
- Maintains a backup copy of (duplexes) log data that is in interim storage for recovery. Log data in interim storage is vulnerable to loss due to system or sysplex failure because it has not yet been hardened to DASD log data sets. System logger duplexes interim storage log data for both coupling facility and DASD-only log streams.
- Produces SMF record type 88 for system logger accounting on a single system. Record type 88 focuses on the usage of interim storage (coupling facility or local storage buffers) and log stream data for a system in the sysplex. Using the record can help an installation avoid the STRUCTURE FULL or STAGING DATA SET FULL exceptions, and perform other tuning and/or capacity planning analysis.

  See *z/OS MVS System Management Facilities (SMF)* for more information on record type 88 and system logger accounting. Sample program IXGRPT1 in SYS1.SAMPLIB shows an example of producing a report from SMF record type 88.
- Ensures that:
  - When the last connection from a system disconnects from the log stream, all log data written by that system to the log stream is offloaded to DASD log data sets.

    System logger also deletes any staging data sets in use for a system at this time.
  - When the last connection to a coupling facility log stream in the sysplex disconnects, all coupling facility log data is offloaded to DASD log data sets and the coupling facility space is returned to XES for reallocation.
- Provides recovery support in the event of application, system, sysplex, or coupling facility structure failure for coupling facility log streams. (See Recovery Performed for DASD-Only Log Streams in *z/OS MVS Programming: Assembler Services Guide* for information about recovery for DASD-only log streams.)

Overview of authorized system logger services

This topic provides an overview of general information about system logger services, including:

- "Summary of system logger services."
- "Coding a system logger complete exit for IXGBRWSE, IXGWRITE, and IXGDELET" on page 778.
- "Using ENF event code 48 in system logger applications" on page 781.

Summary of system logger services

System logger provides the following set of services:

IXGINVNT

Define and maintain log stream and coupling facility structure information in the LOGR policy dynamically. See 'IXGINVNT: Managing the LOGR Policy' in *z/OS MVS Programming: Assembler Services Guide*.

You can also use the IXCMIAPU utility to specify log stream and structure definitions in the LOGR policy. IXCMIAPU also enables you to request a report of current log stream definitions.

IXGCONN
Connect and disconnect an application to and from a log stream. See the topic on "IXGCONN: Connecting to and disconnecting from a log stream" on page 782.

IXGWRITE
Write user-defined log data to a log stream. See 'IXGWRITE: Writing to a Log Stream' in *z/OS MVS Programming: Assembler Services Guide*.

IXGBRWSE
Browse (read) data from a log stream. See 'IXGBRWSE Requests' in *z/OS MVS Programming: Assembler Services Guide*.

IXGDELET
Delete data from a log stream. See the topic on "IXGDELET: Deleting log blocks from a log stream" on page 788.

IXGIMPRT
Import (write) log blocks to a log stream with a log block identifier and timestamp. See 'IXGIMPRT: Import Log Blocks' in *z/OS MVS Programming: Assembler Services Guide*.

IXGQUERY
Retrieve information from a log stream. See 'IXGQUERY: Get Information About a Log Stream' in *z/OS MVS Programming: Assembler Services Guide*.

IXGOFFLD
Initiate an offload of log data from the coupling facility structure for coupling facility log streams and from local storage buffers for DASD-only log streams to DASD log data sets. See 'IXGOFFLD: Initiate Offload to DASD Log Data Sets' in *z/OS MVS Programming: Assembler Services Guide*.

IXGUPDAT
Modify the UTC time stamp maintained in the control information for a log stream. See 'IXGUPDAT: Modify Log Stream Control Information' in *z/OS MVS Programming: Assembler Services Guide*.

The following services contain parameters for both authorized and unauthorized programs:

- IXGCONN
- IXGBRWSE
- IXGWRITE
- IXGDELET

All other system logger services and their parameters can be used by any program. See *z/OS MVS Programming: Assembler Services Guide* and *z/OS MVS Programming: Assembler Services Reference IAR-XCT* for the general description of the Logger services that can be used by any program. Only the guidance for authorized programs using Logger services is contained in this information.

Coding a system logger complete exit for IXGBRWSE, IXGWRITE, and IXGDELET

Choose MODE=SYNCEXIT to specify that the request be processed synchronously, if possible. If the request cannot be completed synchronously, control returns to the caller with a return and reason code indicating that processing of the request is not complete. When processing of the request completes, the exit routine specified at connect time gets control. (The exit routine is specified on the COMPLETEEXIT parameter on the IXGCONN request.)

You can use the REQDATA parameter with MODE=SYNCEXIT to specify user-defined information relating to the request. For example, you can use REQDATA to specify information about control blocks needed by the complete exit. Your application must be in supervisor state, system key to use the MODE=SYNCEXIT and REQDATA parameters.

When a system logger request cannot be completed synchronously, system logger schedules an SRB to complete processing of the request before it returns control to the caller. While the SRB runs independent of the requesting task, the SRB might encounter an error from which it cannot recover. The SRB ensures that the error condition is percolated to the task that issued the system logger request.

Your complete exit provides a mechanism for system logger services to let you know when your asynchronously processed system logger IXGBRWSE, IXGDELET, or IXGWRITE request completes. You provide the address of your complete exit using the COMPLETEEXIT parameter when you issue the IXGCONN REQUEST=CONNECT macro to connect to the log stream. You will be informed of request completion through your complete exit if you specify MODE=SYNCEXIT and the system processes your request asynchronously.

You must be running in supervisor state and a system key to specify MODE=SYNCEXIT on a IXGBRWSE, IXGDELET, or IXGWRITE system logger service request.

Information passed to the complete exit

When the complete exit gains control, it receives the following information about the system logger request in the complete exit parameter list (CMPL), mapped by the IXGCMPL macro:

CMPLREQDATA
Information passed to the complete exit by the issuer of the system logger service request. The use of this optional field is user-defined.

CMPLRETCODE
Return code from the system logger service request.

CMPLRSNCODE
Reason code from the system logger service request.

CMPLANSAREA@
Answer area address. The answer area is mapped by IXGANSAA macro.

CMPLSTREAMTOKEN
Stream token of the log stream connector.

Environment

The complete exit receives control in the following environment:

Authorization:
Supervisor state, and PSW key 0

Dispatchable unit mode:
SRB

Cross memory mode:
PASN=HASN=SASN. PASN, HASN and SASN are equal to the PASN at the time of the connect to the log stream.

AMODE:
31-bit

ASC mode:
Primary ASC mode

Interrupt status:
Enabled for I/O and external interrupts

Locks: No locks held.

Control parameters:
None.

Input specifications

System logger services pass information to the complete exit in registers.

Registers at entry: When the complete exit receives control, the GPRs contain the following information:

Register	Contents
0	Does not contain any information for use by the complete exit.
1	Address of a fullword containing the address of the CMPL
2-12	Do not contain any information for use by the complete exit.
13	Address of a standard 72-byte save area.
14	Return address to system logger services.
15	Entry point address.

When the complete exit receives control, the ARs contain no information for use by the complete exit.

Return specification: Your exit must return control to the system by branching to the address provided on entry in register 14. There are no requirements for the GPRs or ARs to contain any particular value.

Programming considerations

If you have more than one outstanding system logger request being processed asynchronously, multiple instances of your complete exit might run concurrently as system logger services process your request completions. Therefore, you should consider coding your complete exit as a reentrant program.

You must make storage for any of the keywords representing output areas accessible at the time of the completion exit, such as the following:

Macro	Keywords
IXGBRWSE	ANSAREA, BROWSETOKEN, BUFFER, BUFFER64, ANSAREA, BLKSIZE, TIMESTAMP, RETBLOCKID
IXGDELET	ANSAREA, OBLOCKID
IXGWRITE	ANSAREA, RETBLOCKID, TIMESTAMP

You may not be able to access the input parameter data area while your complete exit is running. If you want to save the parameter information for later processing, make a copy of it before your complete exit returns control to the system.

Consider obtaining separate storage for each logger request to store output fields and copies of input parameters and passing the storage to the completion exit using the REQDATA parameter. The storage should be accessible by the logger requestor and the completion exit.

The return and reason code for the request is stored in ANSAA_ASYNCH_RETCODE and ANSAA_ASYNCH_RSNCODE.

In certain instances, the system must quiesce the activity of user exits in order to perform cleanup processing. The following illustrates scenarios where this processing occurs:

- Connection termination

  When a user disconnects while a MODE=SYNCEXIT request is outstanding, the complete exit will not be called.

  If the connecting task terminates, the system will issue a PURGEDQ against SRB that are associated with the connection. Since all complete exit SRBs are associated with the connecting task's TCB, any complete exits that are active when the connecting task terminates could be interrupted with an ABEND x'47B' reason code 0.
- System logger termination

  When the system logger address space terminates, it attempts to inform active requests of logger termination. If the complete exit has not been scheduled, it is scheduled at this time with a return code and reason code indicating that the system logger address space has terminated.

The system logger initializes a recovery environment before it calls the complete exit. Should the complete exit fail and percolate to the system logger's recovery routine, the task that did the corresponding connection to the log stream is abended and retry is not permitted. The abend code will be 1C5 and the abend reason code 00030006.

Using ENF event code 48 in system logger applications

System logger issues ENF event code 48 to broadcast status changes in the system logger address space, log streams, and coupling facility structures. Since these status changes can affect the outcome of system logger service requests, **IBM suggests** that you use ENF event code 48 to receive notification of these status changes, using the ENFREQ service to listen for event 48. Note that your program must be authorized to use the ENFREQ service. Applications should issue the ENFREQ service to listen for event 48 before connecting to a log stream.

For example, suppose an IXGWRITE request fails with a return and reason code indicating that some system logger resource is unavailable, perhaps because the system logger address space has failed or because a structure rebuild is in progress for the coupling facility structure for the log stream. Before the application can resume issuing system logger service requests. it must listen for the event code 48 notification that the resource is available again.

In order to listen for ENF code 48 events, you must code an SRB-type listen exit for event code 48 events to scan the event 48 parameter list for status information on the system logger component, log streams, and coupling facility structures associated with log streams. The listen exit must be in place before system logger applications are activated.

If you use ENF event code 48 to receive information about system logger events, make sure that you take into account the asynchronous nature of the ENF exit. You might get notified of events out of sequence, being notified for instance, that a problem has been resolved before you get a return and reason code describing a problem.

For example, if you issue IXGWRITE to write data to the log stream while the coupling facility structure space allocated for the log stream is full, you might get an ENF 48 notification that the structure is no longer full before you get the return and reason code from IXGWRITE to say that the structure is full.

Applications that do not want to use ENF event code 48 or that are unauthorized and cannot use ENFREQ will still receive logger service return and reason codes indicating failure or resource shortages. These applications can simply set a timer and then retry the requests to see if the problem has resolved itself.

References:

- See “Writing an ENF event 48 listen exit” on page 788 for information on ENF 48 events, and coding your ENF event 48 listen exit.
- See *z/OS MVS Programming: Assembler Services Guide* for guidance about using the ENFREQ macro.
- See *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG* for reference information on the ENFREQ macro.

Figure 85 shows an example of how to issue the ENFREQ service to listen for ENF event code 48 and specify a listen exit to analyze the event 48 parameter list:

```
ENFREQ ACTION=LISTEN,
      CODE=ENFPC048,
      ESTBNME=THISMOD,
      EXITNME=LOGLISTEN,
      SRBEXIT=(R02),
      EOM=YES,
      DTOKEN=ENFREQ_DTOKEN,
      RETCODE=ENFREQ_RETCODE
```

Figure 85. Issuing ENFREQ to Listen for ENF Event Code 48

IXGCONN: Connecting to and disconnecting from a log stream

Use the IXGCONN service to connect to or disconnect from a log stream. An application must issue IXGCONN with REQUEST=CONNECT before it can read, write, or delete data in a log stream.

When the IXGCONN REQUEST=CONNECT request completes, it returns a unique connection identifier, called a **STREAMTOKEN**, to the calling program. The application uses the token in subsequent logger service requests to identify its connection to the log stream.

See *z/OS MVS Programming: Assembler Services Guide* for more details on connecting to a log stream and log stream tokens.

Connecting as a resource manager

A resource manager is an application you can write and associate with a log stream to manage resources and processing for a log stream.

For example, via the resource manager user exit, a resource manager might be notified of a write or delete request issued against a log stream. The resource manager can then perform further processing to accept, reject, or override the delete request with a different log block identifier. A resource manager application can be helpful to perform any management functions on behalf of a log stream.

Before a resource manager can connect to a log stream, the name of the resource manager must be specified in the log stream definition in the LOGR couple data set. You can specify **one** resource manager name for a log stream in the log stream definition. See *z/OS MVS Setting Up a Sysplex* for setting up a log stream definition in the LOGR couple data set.

If you specify a resource manager name for a log stream in the LOGR policy, the resource manager specified must connect to the log stream. If the resource manager does not connect, system logger will not process any IXGDELET requests to delete log data. This is so that the resource manager will not miss any information about deletes issued against the log stream.

The resource manager connects to the log stream it manages using the RMNAME, RMEXIT, RMDATA, and RMEVENTS parameters on the IXGCONN service. You must be running in supervisor state and a system key to use these parameters. The connect request must be issued from the resource manager address space. The resource manager address space must be non-swappable with an authorization index (AX) value of 1, or all invocations of the resource manager exit will fail.

Note that only one resource manager can connect to a log stream from a given system. The resource manager **can** connect to multiple log streams.

Use the resource manager parameters as follows:

RMNAME

Specifies the name of the resource manager program connecting to the log stream. This is the same name specified on the RMNAME parameter in the LOGR couple data set log stream definition.

RMEXIT

Specifies the name of a resource manager user exit. The resource manager exit is called when write or delete requests (as specified on the RMEVENTS parameter) are issued against the log stream that the resource manager manages. The RMEXIT keyword is required with RMNAME. For information on the resource manager exit, see "Coding a resource manager exit for IXGCONN" on page 784.

RMEVENTS

Specifies that write or delete requests issued against the log stream are to trigger the resource manager exit. RMEVENTS is required with RMNAME. You can specify RMEVENTS=LBWRITE, RMEVENTS=LBDELETE, or RMEVENTS=(LBWRITE,LBDELETE).

RMDATA

Specifies user-defined data to the resource manager. This data is then passed to the resource manager user exit when the exit is called.

RMDATA is required with RMNAME.

Using ENF event code 48 with a resource manager

System logger issues many ENF event code 48 of use to a resource manager application.

See:

- "Using ENF event code 48 in system logger applications" on page 781 for using ENF event code 48 events.
- "Writing an ENF event 48 listen exit" on page 788 for ENF event code 48 events and coding an ENF listen exit.

Using ENF event 48 when a connect request is rejected

If a connect request is rejected applications should listen for ENF 48 event informing listeners that the condition has changed or the problem has been resolved. Upon receipt of the ENF 48 event indicating that the problem is resolved, the application can retry the request.

Coding a resource manager exit for IXGCONN

A resource manager exit provides a way for system logger to inform a resource manager of write and delete requests so that the resource manager can perform further processing.

When the resource manager connects to the log stream, it specifies:

- The name of the resource manager on the RMNAME parameter.
- The address of the resource manager exit on the RMEXIT parameter.
- User-defined data on the RMDATA parameter.
- The requests that will trigger the resource manager user exit (write and/or delete requests) on the RMEVENTS parameter.

You must be running in supervisor state and a system key to specify the RMNAME, RMEXIT, RMDATA, or RMEVENTS parameters on the IXGCONN request. The resource manager address space must be non-swappable with an AX value of 1, or all invocations of the resource manager exit will fail.

When a write or delete request occurs against the log stream, system logger gives control to the resource manager exit, passing a parameter list. The resource manager exit runs in the resource manager address space.

The resource manager exit is called as follows:

- For a write request, the resource manager exit is called after the write request completes. If staging data sets are in use for the connection, the exit is called after the write to the staging data set completes.
- For a delete request, the resource manager exit is called before the delete request is processed. This allows the resource manager exit to accept, reject, or override the delete request on behalf of the log stream. See "Overriding delete requests" on page 787.

The resource manager exit is always invoked before the request completion is reported to the system logger application that issued the request.

Information passed to the resource manager exit

When the resource manager exit gains control, it receives the following information in the resource manager exit parameter list (RMEPL) mapped by the IXGRMEPL macro:

RMEPLDELETEREQUEST
If on, indicates that the request that gave the resource manager exit control was a delete request.

RMEPLWRITEREQUEST
If on, indicates that the request that gave the resource manager exit control was a write request.

RMEPLGMTTIMESTAMP
Timestamp obtained immediately prior to calling the resource manager.

RMEPLRMNAME
Name of the resource manager the exit belongs to.

RMEPLRMDATA
Data specified in the RMDATA parameter on the resource manager's IXGCONN request, if specified.

RMEPLIDENTIFICATION
System-unique identification of the connection on whose behalf the exit is being called

RMEPLLOGSTREAMNAME
Name of the log stream associated with the resource manager.

RMEPLBLOCKSALLSPECIFIED
If on, BLOCKS=ALL was specified on the IXGDELETE request

RMEPLBLOCKSRANGESPECIFIED
If on, BLOCKS=RANGE was specified on the IXGDELETE request.

RMEPLFORCESPECIFIED
If on, FORCE=YES was specified on the IXGDELET request. This delete request cannot be overridden by the resource manager.

RMEPLDELETEBLOCKID
The block identifier was specified on a IXGDELET BLOCK=RANGE request.

RMEPLDELETEOVERRIDEBLOCKID
Override block identifier. Resource manager places the override block identifier in this variable to override the block identifier issued on the IXGDELET request. If FORCE=YES was specified on the IXGDELET request, the content of this field is ignored and the block identifier specified on IXGDELET is not overridden.

RMEPLADDEDBYTES
The number of bytes that system logger adds to a user's log block for prefix and suffix information. The prefix and suffix areas are not seen in the copy of the user's buffer presented to the resource manager, but if you add together fields RmeplWriteBlockID, RmeplLogDataLength and RmeplAddedBytes, you can calculate the next block identifier that will be assigned for a log block written to the log stream.

RMEPLLOGDATALENGTH
The number of bytes of user log data specified. This is the BLOCKLEN value specified on the IXGWRITE request.

RMEPLWRITEBUFFERPTR
Pointer to the buffer containing the log data written to the log stream if the RMEPLWRITEREQUEST bit is set on.

RMEPLWRITEBLOCKID
Block identifier assigned to the log block written to the log stream.

RMEPLWRITEGMTTIMESTAMP
Timestamp assigned to the log block written to the log stream.

RMEPLWRITELOCALTIMESTAMP
Local time stamp assigned to the log block written to the log stream.

RMEPL_RMEXIT_WORK_AREA
256 byte work area for use by the resource manager.

Environment

The resource manager exit receives control in the following environment:

Authorization:
Supervisor state with PSW key 0

Dispatchable unit mode:
Task or SRB

Cross memory mode:
PASN=resource manager address space, any HASN, any SASN

AMODE:
31-bit

ASC mode:
Primary ASC mode

Interrupt status:
Enabled for I/O and external interrupts

Locks: No locks held.

Control parameters:
None.

Input specifications

System logger services pass information to the resource manager exit in registers.

Registers at entry

When the resource manager exit receives control, the GPRs contain the following information:

Register	Contents
0	Does not contain any information for use by the resource manager exit.
1	Address of a fullword containing the address of the RMEPL
2-13	Do not contain any information for use by the resource manager exit.
14	Return address to system logger services.
15	Entry point address.

When the resource manager exit receives control, the ARs contain no information for use by the resource manager exit.

Return specification

Your exit must return control to the system by branching to the address provided on entry in register 14. Registers 2-13 must contain the same information at output that they did on input.

Programming considerations

- The resource manager exit is called before write or delete event completion is reported to the unit of work that initiated the IXGWRITE or IXGDELET service request. Any additional processing by the exit that results in thread suspension or affects thread response time should be performed under a different work unit.
- The resource manager exit must be prepared to receive control in either SRB or task mode. The exit should run with an EUT FRR for recovery. While the FRR remains in effect, no SVCs can be issued, no new asynchronous exits are dispatched, and no vector instructions can be executed. See *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO* for information on the SETFRR service.
- All storage obtained by the resource manager exit should be associated with the TCB that owns the cross-memory resources for that address space, whose address is the ASCBXTCB.
- You can access the input parameter data area only while your resource manager exit is running. If you want to save the parameter information for later processing, make a copy of it before your resource manager exit returns control to the system.

Overriding delete requests

Your resource manager can override parameters specified on an IXGDELET request by manipulating the RMEPLDELETEOVERRIDEBLOCKID field mapped by the IXGRMEPL mapping macro. The resource manager manipulates the field in the parameter list (RMEPL) passed to the resource manager exit as follows:

- Proceed with the delete operation as requested on the IXGDELET request. This is specified by placing a return code of binary zeros in register 15.
- Do not proceed with the delete operation requested on the IXGDELET request. This results in no log blocks being marked for deletion in the log stream. This is specified by placing a return code of X'08' in register 15.
- Override the log block identifier specified on the IXGDELET request with one specified by the resource manager. The overriding log block identifier must be less than or equal to the log block identifier specified on the IXGDELET request. This is specified by placing a return code of X'04' in register 15.

If you specify FORCE=YES on a delete request, the resource manager exit is called, but cannot override the delete request.

If you specify AUTODELETE=YES for a log stream and you also manage that log stream with a resource manager and a resource manager exit, note that the automatic deletion processing takes precedence over the delete override processing performed by the resource manager exit. Log data that is deleted by automatic deletion does not trigger the resource manager exit, so the exit cannot override the delete request. IBM recommends AUTODELETE=NO for a log stream managed by a resource manager that needs to override delete requests.

When the resource manager exit hangs

If the resource manager exit hangs, the system logger application that issued the write or delete request that triggered the exit might not be able to complete. Do the following to resolve the hang:

- Cancel the task that issued the write or delete request.
- Cancel the resource manager's address space.
- Restart the application that issued the write or delete request.
- Correct the hang condition in the resource manager exit.

- Restart the resource manager's address space.

If the resource manager abends and percolates the error back to system logger's recovery environment, the resource manager is disabled. When a resource manager is disabled, the exit for the resource manager is no longer called by write or delete requests against the log stream **on the system where the resource manager abended.** An ENF 48 event is issued when the resource manager exit is disabled.

IXGDELET: Deleting log blocks from a log stream

Using the IXGDELET service, you can mark some or all of the log blocks in the log stream for deletion. The IXGDELET service can be invoked from an unauthorized (non-supervisor state, problem program key mode) with the exception of when the MODE=SYNCEXIT option is used as described in topic “Coding a system logger complete exit for IXGBRWSE, IXGWRITE, and IXGDELET” on page 778. Additionally, when a log stream resource manager is active, it runs authorized (supervisor state, system PKM) and can affect the delete requests for the log stream.

Delete requests and resource manager exit processing

If you are using a resource manager exit, your resource manager exit can override certain delete requests (see “Coding a resource manager exit for IXGCONN” on page 784). You can keep the resource manager exit from overriding a delete request by specifying the FORCE=YES parameter. FORCE=NO, which is the default, allows a resource manager exit to override the delete request. If the resource manager overrides a delete request, system logger returns the overridden block identifier in the OBLOCKID output parameter on IXGDELET.

A resource manager program must connect to the a log stream in order to override delete requests. See “IXGCONN: Connecting to and disconnecting from a log stream” on page 782 for more information.

Setting up the system logger configuration

- To set up a system logger configuration for a logging function or application, see the information on planning for system logger functions in *z/OS MVS Setting Up a Sysplex*.
- For system logger applications, **IBM recommends** that you use ENF event code 48 and write an ENF event code 48 listen exit. See “Writing an ENF event 48 listen exit.”

Writing an ENF event 48 listen exit

Before activating system logger applications that write to the log stream, you should set up an exit to listen for and analyze system logger status information broadcast by ENF as event code 48. Note that your application must be supervisor state, system key to set up an ENF listen exit. Event 48 includes status information about log streams, the system logger component, and log stream coupling facility structures. System logger applications can use the ENF event 48 listen exit to monitor status and changes.

Some events are single system in scope, while others are broadcast to all the systems in the sysplex.

Each application registers interest in ENF event code 48 signals using the ENFREQ macro (see “Using ENF event code 48 in system logger applications” on page 781).

When an application is notified through a return and reason code of a problem, such as a coupling facility structure rebuild in progress, system logger address services unavailable, or loss of connectivity to a coupling facility structure, the application can listen for an event 48 signal signifying that the problem has been resolved. Note that a program must be authorized (supervisor state) to use the ENFREQ service.

When the application registers interest in event 48 system logger events, system logger passes the event 48 parameter list containing information about the event to the listen exit. The parameter list is mapped by macro IXGENF, see *z/OS MVS Data Areas* in the z/OS Internet library (http://www.ibm.com/systems/z/os/zos/bkserv/). The parameter list contains the specific code 48 events, the reasons associated with them, specific information for some reasons, and a section of information about log streams and coupling facility structures affected by the event.

When examining the event flag fields in IXGENF, the following information can be used to determine which variation of the IXGENF data is being presented to the Listen Exit.

- If IXGENF field IxgenfLogStreamCount is non-zero, the variation of the IxgenfUnion1 area is an array of IxgenfLogStreamNames structures. The number of elements in this array is the value contained in the IxgenfLogStreamCount field.
- Otherwise, the following table applies:

Table 99. IxgenfUnion1 variation when IXGENF bit set to 1

IXGENF bit set to 1	IxgenfUnion1 variation
IxgenfRMDisabled	IxgenfResMgrDisabled
IxgenfLogStreamDefUpdate	IxgenfInventoryDefUpdate
IxgenfLogStreamDelete	IxgenfInventoryDelete
IxgenfLogStreamConnDisc	IxgenfConnDiscInfo
IxgenfLogStreamOffloadComplete	IxgenfWrOffLoadInfo

The listen exit should **only** be used to analyze the IXGENF parameter list for event code 48 to see whether the particular event applies to the particular application or connector. IBM suggests that the listen exit then communicate with the application about the event and let the application react or take any necessary actions, such as stop issuing services, re-IPL, and so forth. This is recommended because:

- The application or connector can determine whether the event actually affects them before taking any action. All connectors are informed of event 48 events, whether they affect their particular log stream or coupling facility structure or not.
- It ensures that the action needed to address an event will be coordinated with the application program since the listen exit must be in SRB mode while the application is in task mode. If you code the exit to act on event code 48 status events, the exit might tie up system logger resources.

The ENF 48 signals issued by System Logger contain different types of information:

- State of the System Logger system service address space - whether or not IXGxxxxx services should even be issued.

- State of log stream resources, generally they become available or unavailable for various reasons, or when a log stream is defined, updated, deleted, and so forth.
- The IXGENF mapping, which is input to the ENF 48 listen exit routine you establish, is generally organized as follows:

  event
  Main indication of services or resource state change

  even reason
  Incident that occurred causing the state change

  even specific reason
  More details on event reason

The ENF event code 48 events are:

- System logger address space and services have become available.
- System logger address space and services are not available for the life of this IPL.
- Log streams associated with the coupling facility structure specified in this parameter list have become available. The event reasons for this event are:
  - A coupling facility structure rebuild in progress has completed.
  - The coupling facility structure is no longer too full to accommodate more log data.
  - A staging data set full condition has ended.
- Log streams associated with the coupling facility structure specified in this parameter list are not available. The event reasons for this event are:
  - A rebuild has been started for the coupling facility structure.
  - A rebuild for the coupling facility structure has failed. The specific reasons for this condition in the parameter list are:
    - The system lost connectivity to the coupling facility structure.
    - The coupling facility structure failed.
- A change in the status of resources for a log stream has occurred. The event reason for this event is possible loss of log stream data.
- A change in the coupling facility resources available to system logger has occurred. An event reason for this event indicates that ENF event 35 was received to report the change. If the change affects a specific coupling facility structure, the name of the structure is specified in the specific information section of the parameter list.
- A successful connect to or disconnect from a log stream has occurred. The scope of this event is multi-system; each system in the sysplex is notified of this event. The structure version value for DASD-only log streams will be the STCK value for when the log stream staging data set was allocated.
- A log stream definition in the LOGR couple data set has been created. The scope of this event is multi-system; each system in the sysplex is notified of this event.
  - The log stream created is a DASD-only log stream.
    - For DASD-only log streams, an indicator in the parameter is set on (IXGENFINVENTORYDASDONLYYES) and most structure related fields will be set to zero.
- A log stream definition has been deleted from the LOGR couple data set. The scope of this event is multi-system; each system in the sysplex is notified of this event.

- A log stream definition in the LOGR couple data set has been updated. The scope of this event is multi-system; each system in the sysplex is notified of this event.
- System logger completed offload processing for a log stream. All systems with an active connection to the log stream for which the offload was done are notified of the event.
- The resource manager associated with a log stream is disabled because of an abend. The system where the resource manager is disabled is notified.
- The system logger parameter options for the ZAI SERVER/PORT values have changed.

Logger server address space availability considerations

The logger server address space, IXGLOGR, initializes soon after master scheduler initialization (MSI) occurs whenever a z/OS system is IPLed. The address space can be started again after a failure or as a result of certain operator commands. The following are some considerations on the IXGLOGR address space availability:

ENF 48 and system logger initialization

When system logger initializes and is available, the following ENF 48 indicator values appear in the log or on the console?

- IxgenfEvents (IxgenfEventsByte0): IxgEnfSystemLogger Avail

During initialization if the system logger address space abnormally ends, the following values appear in the log or on the console?

- IxgenfEvents (IxgenfEventsByte0): IxgEnfSystemLoggerNotAvailForIPL
  IxgenfEventReasons (IxgenfEventReasonsByte1): IxgEnfLoggerNotAvailXcfLocal

Any subsequent IXGxxx requests result in the following reason code condition:

Return code X'8', reason code X'0890', which indicates that the system logger address space is not available because it has not yet been started or has failed.

If initialization of system logger is occurring but not yet complete, you might receive the following reason code condition:

- Return code X'8', reason code X'0891', which indicates that the system logger address space is not available but is in the process of initializing. The 0891 reason code implies that when system logger eventually becomes available, the system issues the ENF 48 event.

If the system logger server address space is started and later ends either through a catastrophic error or through the FORCE command, the system does not issue an ENF 48 signal. However, any subsequent IXGxxxx service requests receive the reason code condition return code X'8', reason code X'0890' and might also receive the reason code condition return code X'8', reason code X'0891'.

ENF 48 event code scenarios

In the following scenarios system logger has failed to initialize, and you receive the ENF 48 event code. The system waits until the operator restarts system logger.

Assume the following:

1. You make an IXGWRITE request to a log stream and receive a X'0890' or X'0891' reason code that indicates system logger is not available.

2. You can wait for the system to issue an ENF 48 event code and to have the system logger address space restarted.
3. When the system logger address space becomes available, you are no longer connected to the log stream and must issue an IXGCONN connect request to reconnect to the log stream before you can issue the IXGWRITE request.

Assume the following:

1. You issue an IXGCONN request to connect to a log stream and then issue IXGWRITE requests to the log stream that complete processing.
2. The system logger address space then fails, and the operator restarts system logger again.
3. You issue another IXGWRITE request before you process the ENF 48 event code. However, the request fails with either IxgRsnCodeExpiredStmToken (return code X'8'; reason code X'082D'?) or IxgRsnCodeBadStmToken (return code X'8'; reason code X'0806'?) errors.
4. Because the log stream connect token is no longer valid, you must re-connect to the log stream before you can write to it. See “Considerations for logger log stream disconnected” on page 794.

Considerations for logger resources temporarily unavailable

Table 100 includes logger service reason codes that indicate a "resource temporarily unavailable" condition (see IXGCON macro) and identifies the related logger ENF 48 events (see IXGENF macro) that occur for these conditions or resource state changes.

Table 100. Logger services with "temporary unavailable type reason codes" and expected related ENF 48 events

Reason Code	Hex Value	Logger Services	IXGENF event(s)	IXGENF event reasons
IxgRsnCodeCFLogStreamStorFull	860	IXGIMPRT, IXGWRITE	IxgenfLogstreamsAvailable	IxgenfLogstreamStorageAvailable
IxgRsnCodeRebuildInProgress	861	IXGBRWSE, IXGDELET, IXGIMPRT, IXGOFFLD, IXGQUERY, IXGUPDAT, IXGWRITE	• IxgenfLogstreamsNotAvailable, then • IxgenfLogstreamsAvailable	• IxgenfStrRebuildStart • IxgenfStrRebuildComplete or IxgenfStrRebuildFailed
IxgRsnCodeXESPurge	862	IXGBRWSE, IXGDELET, IXGIMPRT, IXGOFFLD, IXGQUERY, IXGUPDAT, IXGWRITE	• IxgenfLogstreamsNotAvailable, then • IxgenfLogstreamsAvailable	• IxgenfStrRebuildStart • IxgenfStrRebuildComplete or IxgenfStrRebuildFailed
IxgRsnCodeStructureFailed	863	IXGBRWSE, IXGCONN, IXGDELET, IXGIMPRT, IXGOFFLD, IXGQUERY, IXGUPDAT, IXGWRITE	• IxgenfLogstreamsNotAvailable, then • IxgenfLogstreamsAvailable	• IxgenfStrRebuildStart • IxgenfStrRebuildComplete or IxgenfStrRebuildFailed
IxgRsnCodeNoConnectivity	864	IXGBRWSE, IXGCONN, IXGDELET, IXGIMPRT, IXGOFFLD, IXGQUERY, IXGUPDAT, IXGWRITE	• IxgenfLogstreamsNotAvailable, then • IxgenfLogstreamsAvailable	• IxgenfStrRebuildStart • IxgenfStrRebuildComplete or IxgenfStrRebuildFailed
IxgRsnCodeStagingDsFull	865	IXGIMPRT, IXGWRITE	IxgenfLogstreamsAvailable	IxgenfStagingDsStorageAvailable
IxgRsnCodeStructureFull	866	IXGCONN	IxgenfLogstreamsAvailable	IxgenfLogstreamStorageAvailable
IxgRsnCodeLocalBufferFull	867	IXGIMPRT, IXGWRITE	IxgenfLogstreamsAvailable	IxgenfLogstreamStorageAvailable
IxgRsnCodeStagingDsFormat	868	IXGIMPRT, IXGWRITE	IxgenfLogstreamsAvailable	IxgenfLogstreamStorageAvailable
IxgRsnCodeDsDirectoryFull	85C	IXGIMPRT, IXGWRITE	IxgenfLogstreamsAvailable	IxgenfLogstreamStorageAvailable
IxgRsnCodeWowError	85D	IXGIMPRT, IXGWRITE	IxgenfLogstreamsAvailable	IxgenfLogstreamStorageAvailable
IxgRsnCodeAddrSpaceInitializing	891	IXGBRWSE, IXGCONN, IXGDELET, IXGIMPRT, IXGINVNT, IXGOFFLD, IXGQUERY, IXGUPDAT, IXGWRITE	IxgenfSystemLoggerAvail	n/a

Table 100. Logger services with "temporary unavailable type reason codes" and expected related ENF 48 events (continued)

Reason Code	Hex Value	Logger Services	IXGENF event(s)	IXGENF event reasons
IxgRsnCodeConnectRebuild	406	IXGCONN	• IxgenfLogstreamsNotAvailable, then • IxgenfLogstreamsAvailable	• IxgenfStrRebuildStart • IxgenfStrRebuildComplete or IxgenfStrRebuildFailed
IxgRsnCodeStructureNotAvail	8B0	IXGCONN	IxgenfSystemLoggerResourceCheg	IxgenfCfResourceChange

The following conditions can result from a coupling facility structure rebuild:

- In Table 100 on page 792 the reason codes indicate that the log stream access is inhibited and the log stream is temporarily unavailable during the structure rebuild.
- When the CF structure rebuild completes, system logger issues an ENF 48 indicates whether the log stream resource is available for use. If the rebuild does not complete successfully, the system issues a new ENF 48 event that indicates the log stream resource is not available for a new reason.
- Initially, system logger issues an ENF 48 event when a structure rebuild starts, indicating the affected log streams are unavailable. The following are ENF 48 indicators that are not available:

ENF 48 indicators that are not available

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamsNotAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte0):	IxgEnfStrRebuildStart
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	

- Normally, when the rebuild completes, the system issues an ENF event to inform the connectors that the log stream is now available. The following are ENF 48 event indicators that are available:

ENF 48 event indicators that are available

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamsAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte0):	IxgEnfStrRebuildComplete
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	

- If the rebuild processing does not complete successfully, the system issues a new ENF 48 event to inform the connectors that the log stream state is not available for a new reason code to be issued. The following ENF 48 event indicators are not available:

ENF 48 event indicators that are not available

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamsNotAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte0):	IxgEnfStrRebuildFailed
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	IxgEnfStrRebuildFailLossConn or IxgEnfStrRebuildFailsStrFail

- For the rebuild failure case, some other action needs to occur to cause the log stream status to change again. The log stream might be disconnected or a subsequent rebuild might succeed. The system issues ENF 48 events for these activities as they occur for the log stream.

  If a rebuild initiated due to a loss of connectivity fails, system logger service requests to exploit affected log streams will result in IxgRsnCodeNoConnectivity.

You may need to change the log stream status again at this point. See the log for related system messages (such as IXG101I and IXG107I), which will provide advice on the appropriate actions.

- IxgRsnCodeStagingDsFormat indicates system logger is formatting the staging data set to be used to duplex the log data that is into the log stream interim storage. This condition is most likely to occur directly after a log stream connection on a system that resulted in the staging data set being newly allocated. Applications should not resume issuing write requests until receiving the ENF event indicating that the log stream (and staging data set) resource is available. The following ENF 48 event indicators are available:

ENF 48 event indicators that are available		
IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamsAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte0):	IxgEnfStagingDsStorageAvailable
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	

- IxgRsnCodeDsDirectoryFull and IxgRsnCodeWowError (85C and 85D) indicates the following:
  - Indicates that the interim storage for the log stream is full and system logger was unable to offload log data to DASD (that is, the log stream data set directory is full or some other offload error occurred). System logger will re-drive its offload attempts for these conditions, which are applicable to both coupling facility structure and DASD-only type log streams.
  - If system logger is able to offload log data, then an ENF event will be issued informing the connectors that the log stream should be available for writing more log data (see Figure 81 on page 773).
  - You can retry your IXGWRITE (or IXGIMPRT) request periodically or wait for the ENF signal that the log stream is available, or disconnect from this log stream and connect to another log stream.
  - For additional actions related to log stream offloads, refer to system messages IXG257I, IXG261E, IXG262A and IXG301I.
- For IxgRsnCodeCFLogStreamStorFull (860) or IxgRsnCodeWowError (85D) on IXGWRITE requests, while waiting for IxgenfLogstreamsAvailable event with IxgenfLogstreamStorageAvailable reason, authorized application can also retry the request after a short wait. When the structure becomes full, system logger starts offloading, and system logger produces the ENF signal when the offload processing is able to relieve the full condition. For these two log stream structure full-type conditions, system logger allows authorized IXGWRITE requests (with exception for MODE=ASYNCNORESPONSE type requests) to be attempted as the structure offloads, and these requests may complete successfully before the ENF signal is received.

Considerations for logger log stream disconnected

See 'System Logger Processing at Disconnection' in *z/OS MVS Programming: Assembler Services Guide* for additional details on log stream disconnect processing and log stream tokens.

Log stream disconnects can occur for a variety of reasons;

- An application explicitly or implicitly disconnects from a log stream
- The result of a CFRM policy
- System environmental conditions
- From internal (system logger component) errors.

System logger will issue an ENF 48 signal for most of the conditions listed above. The IXGENF mapping, which is input to the ENF 48 listen exit routine is organized as follows:

event
Main indication of services or resource state change

event reason
Incident that occurred causing the state change

event specific reason
More details on event reason

However, note that there is a special case for the ENF 48 events when the "event specific reason" actually designates the significant state of the log stream resource. That is when a log stream is disconnected and deemed unavailable and an ENF 48 event with IxgEnfLogstreamsNotAvailable and IxgEnfLogstreamDisconnected are both set on.

The following are ENF 48 events that relate to a log stream being disconnected:

- The log stream connector disconnects from the log stream by issuing an IXGCONN REQUEST=DISCONNECT,STREAMTOKEN value.

  System logger will issue an ENF 48 event that indicates that a connector has disconnected from the log stream.

ENF 48 event indicators:

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamConnDisc

  IxgenfConnDiscInfo section will identify the system and log stream name for the disconnect event along with some additional indicators:

  – IxgEnfConnDiscFlags

  – IxgenfConnDiscDisConnect

- For structure-based log streams, system logger can disconnect a requestor when certain structure failure conditions occurs. An ENF 48 event is issued informing the connectors that they have been disconnected (CFRM policy action), and any future system logger service requests using the same log stream token received the return code 8, reason code 82D response.

ENF 48 event indicators:

IxgenfEvents	(IxgenfEventsByte0):	IxgEfLogstreamsNotAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte0):	IxgEnfXESRecommendAction
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	IxgEnfLogstreamDisconnected and IxgEnfLossOfConnectivity

- When system logger is unable to allocate a new **staging data set** when an I/O or access error occurs for DASD-only logstreams the log stream is automatically disconnected.

  System logger issues an ENF 48 informing the connectors that they have been disconnected (staging data set error), after which any system logger that uses the same log stream token receives the return code 8, reason code 82D response.

ENF 48 event indicators:

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamsNotAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte0):	IxgEnfReqLogResNotAvail
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	IxgEnfLogstreamDisconnected and IxgEnfStgAllocErr

- System logger disconnects all connectors on the target system as a result of an operator SETLOGR FORCE,DISConnect,LSN=*logstremname* command. System logger issues an ENF 48 informing all the connectors that they have been disconnected (operator command), after which any system logger that uses the same log stream token receives the return code 8, reason code 82D response.

ENF 48 event indicators:

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamsNotAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte1):	IxgEnfSetLogrForceDisconnect
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	IxgEnfLogStreamDisconnected

- When the system logger server address space terminates and is restarted while the log stream connector is persistent (for example, when application address space and connecting task remained intact) then any subsequent use of a log stream token obtained before system logger terminated would be considered expired. See "Logger server address space availability considerations" on page 791 for more details pertaining to this condition.

  When system logger becomes available again, system logger will issue an ENF 48 and the application should connect to the log stream and not make use of the expired stream token.

ENF 48 event indicators:

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfSystemLoggerAvail

- There are also **severe** error conditions that can occur within the system logger component that might cause the log stream to be disconnected unexpectedly.

  An ENF 48 event is issued informing the connectors that they have been disconnected (component error) and any future Logger service requests using the same log stream token will receive the rc8,rsn82D response.

ENF 48 event indicators:

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogstreamsNotAvailable
IxgenfEventReasons	(IxgenfEventReasonsByte1):	IxgEnfComponentError
IxgenfEventSpecificInfo	(IxgenfEventSpecificInfoByte0):	IxgEnfLogstreamDisconnected

When system logger completes the disconnection processing for all connectors on a system and the log stream will no longer appear "actively" connected on that system, the system issues an ENF 48 event. For example, if a DISPLAY LOGGER,CONN,LSN=*logstreamname* command is issued after the ENF 48 event, the resulting IXG601I message indicates the percent of the connectors to the log stream for the log stream as zero (num_conn =0).

The ENF 48 event indicates that it is a "system level" log stream disconnect along with the number of systems in the sysplex still connected to this log stream.

ENF 48 event indicators:

IxgenfEvents	(IxgenfEventsByte0):	IxgEnfLogStreamConnDisc
IxgenfEventReasons	(IxgenfEventReasonsByte1):	IxgEnfSystemLevelDisc

IxgEnfConnDiscInfo section identifies the system and log stream name for the disconnect event along with some additional indicators:

ENF 48 event indicators:

IxgenfConnDiscFlags	IxgEnfConnDiscDisConnect
IxgenfConnDiscCount	Contains the number of systems still connected to this log stream in the sysplex.

Note: A common problem in the early coding is that a non-persistent task will be used in the log stream connecting address space. The address space establishes an initialization task to obtain resources. It stores the log stream token in persistent storage, but then the task terminates. For this situation, Logger will automatically disconnect from the log stream since the "owning" task terminated (via task termination resource manager operation). The correction for this is to simply connect to the log stream in an address space persistent task (at least keep the task persistent for as long as the connection, that the STREAMTOKEN, is expected to be kept for use). Logger will **not** issue an ENF 48 event for this specific condition. However, the "system level" disconnect event described above will be issued.

When things go wrong: Recovery scenarios for system logger

This information describes some of the failures that can affect system logger applications and the action taken by system logger in response. Only the scenarios specific to authorized programming environments are described here. See *z/OS MVS Programming: Assembler Services Guide* for the recovery scenarios that pertain to both authorized and unauthorized programs.

When a resource manager fails

When a resource manager percolates to the recovery environment of system logger, it is disabled with abend X'x22', regardless of whether it had the opportunity to retry. When this happens, the resource manager must disconnect from the log stream and then reconnect in order to activate the resource manager exit.

When a resource manager is disabled, an ENF 48 event is issued on the system where it is disabled. If an SDWA is available to system logger's recovery routine, the abend code is included in the ENF parameter list mapped by macro IXGENF.

The resource manager should examine the abend code and decide whether to disconnect and terminate or to disconnect and reconnect to the log stream.

See “Writing an ENF event 48 listen exit” on page 788 for information on ENF 48 events.

Chapter 31. System REXX

System REXX is a z/OS component that allows REXX execs to be executed outside of conventional TSO/E and Batch environments. REXX has long been considered one of the fastest development languages for system exit and utilities work on z/OS. The possibilities for exploiting REXX code through the use of System REXX are vast, whether to provide operator assists or to provide an easy way to process files and strings. The System REXX environment provides a function package that allows a REXX exec to invoke system commands and to return results back to the invoker in a variety of ways. System REXX execs may be initiated through an assembler macro interface called AXREXX or through an operator command.

There are two different System REXX environments supported:

- TSO=NO
- TSO=YES

In both environments, the exec runs in problem state, key 8, in an APF authorized address space. Any modules that are loaded, linked or attached from the exec, must reside in an APF authorized library; otherwise, a X'306' abend occurs. In both cases, the REXX exec runs under the enclave of the AXREXX invoker when the invoker can be classified; otherwise, the exec runs under the enclave of AXR.

There can be up to 64 REXX worker tasks running TSO=NO execs and up to 8 TSO server address spaces running TSO=YES execs. If a worker task is not available for an inbound TSO=NO request, or if a TSO server address space is not available for a TSO=YES request, the request is queued and the requestor is suspended if SYNC=YES is specified. The order in which System REXX processes queued requests may not be in the same order that the requests have been submitted. AXREXX invokers that use SYNC=YES should consider the potentially long wait time.

At most 5000 active and waiting requests are allowed to exist at any time. When this threshold is reached, subsequent AXREXX requests are rejected until the number of active and waiting requests drops to 4000. ENF signals (65) are issued when the threshold is exceeded, when the total number of requests is getting close to the threshold, and when acceptance of inbound requests is resumed.

TSO=NO environment: When TSO=NO is specified on the AXREXX invocation, the exec is executed in an MVS host command environment, sharing the address space where it is executing with up to 63 other concurrently running TSO=NO execs. In addition to MVS, the following host command environments are supported:

- APPCMVS
- ATTACH
- ATTCHMVS
- ATTCHPGM
- BCPii
- CPICOMM
- LINK
- LINKMVS
- LU62

Data set allocation other than provided by the AXREXX macro is not supported in this environment. Applications that perform input/output to data sets other than those specified on the REXXINDSN and REXXOUTDSN AXREXX keywords must use TSO=YES.

TSO=YES environment: The TSO=YES environment supports all of the host commands listed under the TSO=NO environment, along with some additional host commands supported by TSO. If you specify TSO=YES on the AXREXX invocation, the exec will run isolated in a single address space, and can safely allocate data sets without concern of a DDNAME conflict with a concurrently running exec. If the exec exits with data sets allocated, System REXX will free the allocations. The TSO=YES environment is established with the Terminal Monitor Program (TMP) when AXRRXWKD is specified as an authorized command in IKTJTSOxx, otherwise the TSO/E Environment Service is used. When established using the TMP, the TSO=YES environment supports CONSOLE host commands, the SUBMIT command, and several other foreground initiated background commands that are unavailable with the TSO/E Environment Service. If the exec is initiated when the primary subsystem is not active, the exec runs under the MASTER subsystem which restricts available TSO commands, regardless of whether the TMP is used. Only the TSO=YES environment supports SYSCALL (z/OS UNIX) host commands that are only available to requests associated with RACF user ID's that have OMVS segments defined to them, which establishes the level of z/OS UNIX authorization. In particular, if execs are initiated from the operator console, the operator must be logged on for many SYSCALL host commands to work.

The following list shows additional TSO host commands that are supported in a TSO=YES environment:

- ALLOCATE (except for the SYSOUT operand)
- ATTLIB
- ATTRIB
- CALL
- DELETE
- EXEC
- FREE
- HELP
- OUTTRAP
- PROFILE
- RECEIVE
- RMM
- SEND
- SMCOPY
- TIME
- TRANSMIT

Note: See *z/OS DFSMSrmm Managing and Using Removable Media* for details of the considerations for use of the RMM TSO subcommands.

No other TSO Services and facilities are supported. In particular, when non-supported host commands are invoked, the following errors might be encountered:

- Some authorized host commands may not work under the TSO Environment Service. Starting System REXX under the TMP should resolve this.
- The PSCBUSER field in the PSCB can be * for a TSO=YES environment when the security environment specified on the AXREXX invocation is not associated with a valid userid. When this occurs, the value is returned by the userid() function in REXX, and can cause problems for TSO/E commands or services that require a valid user ID.
- JES facilities are available when the primary subsystem is active. When the primary subsystem is not active, the exec is run under the MASTER subsystem.

When a TSO=YES exec is dispatched, it receives control with the following TSO/E profile attributes set:

- NOPREFIX
- NOCHAR
- NOLINE
- NOPROMT
- NOINTERCOM
- NOPAUSE
- MSGID
- NOMODE
- WTPMSG
- NORECOVER
- PLANGUAGE(ENU)
- SLANGUAGE(ENU)
- VARSTORAGE(HIGH).

Other restrictions that users should be aware of include:

- System REXX does not support the establishment of additional Language Processor Environments using any of the TSO/E REXX Customizing Services (IRXINIT or IRXTERM for example). In addition, when a REXX exec invokes an assembler program (by LINKMVS or other means), the contents of field ENVBLOCK_USERFIELD in the IRXENVB data area should not be altered in any way. If the program needs to maintain a storage area across calls, it must use the name/token services. See 'Sharing Application Data (Name/Token Callable Services)' in *z/OS MVS Programming: Assembler Services Guide*.
- The STORAGE external function can only be used to read storage.
- The BCPii host command environment does not support being invoked in an exec via the MODIFY AXR command.

Planning to use system REXX

System REXX starts automatically during Master Scheduler Initialization and should run in the SYSSTC service class. There is a SYS1.PARMLIB member (CTIAXR00) for controlling Component Trace (Errors are traced by default) and also a SYS1.SAMPLIB member (AXR00) that may be tailored and copied into SYS1.PARMLIB to override IBM supplied defaults. Additionally, IEASYSnn supports a parmlib concatenation of AXRnn members. For more information, see *z/OS MVS Initialization and Tuning Reference*. If no alternative AXRnn member has been configured and AXR00 is not found, REXX&SYSCLONE is the default value assigned to CPF and SYS1.SAXREXEC the default value assigned to REXXLIB.

The System REXX address space (AXR) and eight TSO Server address spaces (AXR01-AXR08) in which TSO=YES requests are processed must have user IDs defined and be connected to the installation's designated RACF group for system address spaces. This RACF group must have READ authority to the SYS1.PARMLIB and SYS1.SAXREXEC data sets. You must ensure that there are matching entries in either the started procedures table (ICHRIN03) or that a STARTED class profile has been defined that matches each new address space name. If you prefer, both the started procedures table and STARTED class profile might be in place. This action ensures that the correct user ID and system group characteristics are assigned.

The following RACF commands can be used to accomplish this simply and dynamically. If SYS1 is not the RACF group name, substitute it with the correct group name.

```
ADDUSER AXR     DFLTGRP(SYS1)
ADDUSER AXR01   DFLTGRP(SYS1)
ADDUSER AXR02   DFLTGRP(SYS1)
ADDUSER AXR03   DFLTGRP(SYS1)
ADDUSER AXR04   DFLTGRP(SYS1)
ADDUSER AXR05   DFLTGRP(SYS1)
ADDUSER AXR06   DFLTGRP(SYS1)
ADDUSER AXR07   DFLTGRP(SYS1)
ADDUSER AXR08   DFLTGRP(SYS1)

RDEFINE STARTED AXR*.* STDATA(USER(=MEMBER),GROUP(SYS1))

SETROPTS RACLIST(STARTED) GENERIC(STARTED) REFRESH
```

For details, see the topic "Using Started Procedures" in the *z/OS Security Server RACF Security Administrator's Guide*.

Alternatively, after you substitute installation preferences for user ID and group name, add the following statements to ICHRIN03 which must then be recompiled and link-edited. Also, recognize that an IPL is required for the updated started procedures table to take effect.

Below are entries to be added to ICHRIN03:

```
*
DC CL8'AXR' PROCEDURE NAME
DC CL8'AXR' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
DC CL8'AXR01' PROCEDURE NAME
DC CL8'AXR01' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
DC CL8'AXR02' PROCEDURE NAME
DC CL8'AXR02' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
DC CL8'AXR03' PROCEDURE NAME
DC CL8'AXR03' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
```

```
DC CL8'AXR04' PROCEDURE NAME
DC CL8'AXR04' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
DC CL8'AXR05' PROCEDURE NAME
DC CL8'AXR05' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
DC CL8'AXR06' PROCEDURE NAME
DC CL8'AXR06' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
DC CL8'AXR07' PROCEDURE NAME
DC CL8'AXR07' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
*
DC CL8'AXR08' PROCEDURE NAME
DC CL8'AXR08' USERID
DC CL8'SYS1' GROUP NAME
DC XL1'00' TRUSTED ATTRIBUTE BIT
DC XL7'00' RESERVED
```

For details, see the topic "The Started Procedures Table (ICHRIN03)" in the *z/OS Security Server RACF System Programmer's Guide*. A full example of ICHRIN03 can be found in SYS1.SAMPLIB.

The System REXX address space (AXR) and its TSO Server address spaces should run in the SYSSTC service class and should not be explicitly classified to a different service class. The service class can be displayed by issuing DISPLAY JOBS,AXR from the operator console.

The System REXX address space, AXR, is non-cancelable. If absolutely necessary, you can terminate the AXR address space by issuing the FORCE AXR,ARM command. Using the FORCE command can have ramifications for your system, so read FORCE command in *z/OS MVS System Commands* before using it.

The ENF signals that SYSREXX issues when the AXR address starts and ends are listed in “ENF event codes and meanings” on page 168. The operator can restart AXR by using the AXRPSTRT procedure, found in SYS1.PROCLIB. The syntax for restarting AXR can be one of the following:

```
START AXRPSTRT
START AXRPSTRT,AXR=aa
START AXRPSTRT,AXR=(aa,bb,...)
```

where aa and bb are AXRnn parmlib members in SYS1.PARMLIB. If no parmlib members are specified, values from AXR00 are applied if it exists; otherwise, default values are assigned.

You can use the command prefix (CPF) defined in AXR00 to invoke a REXX exec or the SYSREXX STATUS command from a console rather than specifying the MODIFY AXR command. The installation has the option of defining the prefix to be SYSTEM or SYSPLEX in scope. See *z/OS MVS Planning: Operations* for more details about CPF processing. The MODIFY AXR command is discussed in *z/OS MVS System Commands*.

The installation should:

- Evaluate the security issues involved with System REXX, including who should have access to the data sets in the REXXLIB concatenation.
- Determine who should be allowed to issue MODIFY AXR to submit an exec for execution or to display status from an operator console.

The resource entity for the MODIFY AXR,<exec name> command is MVS.SYSREXX.EXECUTE.<exec name> and the resource entity for the SYSREXX STATUS command is MVS.SYSREXX.STATUS. Both profiles are defined under the OPERCMDS class. For the MODIFY AXR,SYSREXX STATUS command, if SAF cannot decide whether the request is authorized, it will be allowed. For the MODIFY AXR,<exec name> command, if SAF cannot decide whether the request is authorized, it will be allowed only if the invoking console has MASTER authority.

The AXRUSER() keyword parameter in SYS1.PARMLIB(AXRnn) is optional. No default is assigned to this keyword if an AXRnn member is omitted from SYS1.PARMLIB. When AXRUSER(siteuserid) is added to SYS1.PARMLIB(AXRnn), it signifies to System REXX that the security token for siteuserid can be used by the invoker when the AXREXX macro is coded with SECURITY=BYAXRUSER. The siteuserid must already be defined to RACF. System REXX uses the RACF SURROGAT class resource SYSREXX.siteuserid to verify that siteuserid is authorized to perform this role and will subsequently extract the security token associated with siteuserid. This security token is then subsequently used to set up the security environment for any requests specifying SECURITY=BYAXRUSER. This process is necessary for requests that are initiated under the MASTER address space and other address spaces that do not have bonafide security environments. An example of this is an MPF exit invoking a REXX exec in response to a particular message. This process differs from the way JES uses the SURROGAT class in which the submission of a job could be performed on behalf of another user.

Any defined user ID can be selected by specifying AXRUSER(siteuserid) in SYS1.PARMLIB(AXRnn) where siteuserid is the user ID chosen by the installation to be used as the surrogate user ID. The only access requirement for the surrogate user ID is READ authority to SYS1.SAXREXEC and to any data sets included in the REXXLIB concatenation, unless the installation has imposed some access requirements of its own. Use of the surrogate user ID can be further restricted through universal access control and adding just the surrogate user ID to the permission list.

The following RACF commands are used to establish a surrogate user environment for System REXX. Substitute your installation preferences for siteuserid and sitegroupid.

```
ADDUSER siteuserid DFLTGRP(sitegroupid)
RDEFINE SURROGAT SYSREXX.siteuserid UACC(NONE)
PERMIT SYSREXX.siteuserid CL(SURROGAT) ID(siteuserid) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

For example, if AXRSROGT is the preferred siteuserid specification and SYS1 is the RACF group name it is connected to, the following commands are entered by the RACF Security Administrator to complete the setup.

```
ADDUSER axrsrogt DFLTGRP(sys1)
RDEFINE SURROGAT SYSREXX.axrsrogt UACC(NONE)
PERMIT SYSREXX.axrsrogt CL(SURROGAT) ID(axrsrogt) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

For more information, see *z/OS Security Server RACF Command Language Reference*.

The parmlib member AXRnn provides the user with the ability to specify a set (concatenation) of data sets from which SYSREXX attempts to fetch execs. This set of data sets is referred to as the REXXLIB concatenation. SYS1.SAXREXEC contains execs that IBM provides and in general should not be modified. When SYS1.SAXREXEC is not specified in the REXXLIB concatenation, it is appended to the end. For details about AXRnn processing, see *z/OS MVS Initialization and Tuning Reference*. Any execs that are added to the concatenation cannot start with the letters A through I that are reserved for IBM execs.

Security

The SECURITY and UTOKEN parameters on the AXREXX macro determine the security environment that the exec runs in. If omitted, the exec will run under the same security environment as its invoker. The security environment determines the data sets that may be accessed and the commands and programs that may be invoked.

- When SECURITY=BYUTOKEN is specified, the invoker can provide a UTOKEN to define the specific security environment under which the exec should run (see *z/OS Security Server RACROUTE Macro Reference*). By default the AXREXX macro invocation uses SECURITY=BYUTOKEN. In this situation a security token always accompanies the request. It is used to establish the security environment of the invoker under which the exec will run and is limited to the permissions that are applicable to the invoker. Additionally, if the REXX exec uses the AXRCMD function to issue a system command, the security token will be passed to MGCRE ensuring the identical security information is used. The security token value can be explicitly provided using the optional parameter UTOKEN, on the AXREXX macro. If the value is omitted, it will be implicitly set to that of the invoker's security environment.
- When SECURITY=BYAXRUSER is specified, the exec will run under the security environment associated with the value (siteuserid) of the AXRUSER parameter specified in the AXR00 parmlib member. This could be useful if the installation wants to invoke AXREXX in an address space that does not have a security environment such as the MASTER address space.

The exec should not invoke any services that alter the security environment of the task running the exec.

If an exec is invoked from a console that is not logged on, or via the AXREXX macro from an address space that does not have a valid security environment (using SECURITY=BYUTOKEN without specifying a UTOKEN), certain facilities such as SYSCALL host commands will not function properly.

Argument and variable processing

The AXREXX macro allows the invoker to specify up to 20 arguments and 256 variables by specifying the REXXARGS or REXXVARS parameter respectively. To use the REXXARGS and REXXVARS parameters, the AXREXX invoker must create a header section mapped by AXRARGLST followed in contiguous storage by one or more AXRARGENTRY sections. For the REXXARGS parameter, the entries mapped by AXRARGENTRY must appear in the same order as the arguments specified on the ARG statement in the REXX program. The mapping for AXRARGLST and AXRARGENTRY can be found in AXRZARG.

AXRARGLST contains the following:

- AxrArgLstId - Set this to either AxrArgLstAcro or AxrVarLstAcro depending on whether this is for the RexxArgs parameter or the RexxVars parameter
- AxrArgLstVer - Set to 0 (the current version)
- AXRARGLstNumber - Set to the number of Arguments or variables (for example, the number of AXRARGLstEntry's that follow)
- Other fields must be cleared to 0.

AXRARGEntry contains the following:

- AXRARGAddr - Set this to the 64 bit address of the buffer containing the argument or variable. If the argument or variable resides below 2 gig, use AXRARGAddrLow and make sure AXRARGAddrHigh is 0.
- AxrNameAddr - Set this to the 64 bit address of the buffer containing the name of the argument or variable. This field can be set to 0 if this is for an input only argument. If this name resides below 2 gig, use AXRNameAddrLow and set AXRNameAddrHigh to 0.
- AXRARGLength - Set this to the length of the buffer containing the argument or variable. Note that different argument/variable types have specific requirements regarding lengths.
- AXRARGAlet - Set this to the alet of the argument/variable. It must be a public entry on the DUAL of the task that invokes AXREXX. If the argument/vairable resides in the invoker's primary address space, set this to 0.
- AXRARGNameAlet - Set this to the alet of the buffer containing the name of the argument/vairable. It must be a public entry on the DUAL of the task that invokes AXREXX. If the name resides in the invoker's primary address space, set this to 0.
- AXRArgOutLength - System REXX sets this to the length of data returned to the invoker. Note that this value is in units of bytes for types Signed, Unsigned and Char, in units of hex digits (half bytes) for type HexString and in units of bits for type BitString.
- AxrArgNameLength - Set this to the length of the name of the argument. This must contain the actual length of the name and not include any trailing blanks.
- AxrArgType - Set this to the type of the argument/variable.
- AXRARGInput - Set this if the argument/variable in the REXX exec is to be initialized to a value on entry to the exec.
- AXRArgOutput - Set this if you want to retrieve the final value of the argument/variable on exit from the exec for a SYNC=YES request. If the variable is not set by the exec, System REXX will fail the request.
- Other fields must be cleared to zero.

Since the only data type in REXX is the character string, System REXX must first convert input arguments or variables into this format. The invoker must specify the data type of the argument or variable in AXRARGTYPE. The following data types are supported:

1. AXRARGTYPEUNSIGNED - The input is treated as an unsigned integer value. The length must be 4 or 8 bytes.
2. AXRARGTYPESIGNED - The input is treated as 2s complement signed integer value. The length must be 4 or 8 bytes.

3. AXRARGYTPECHAR - The input is treated as a character string. The length can be from 0 to 512 bytes.
4. AXRARGTYPEHEXSTRING - The input is treated as a hexadecimal string. The length is specified in hexadecimal digits (2 per byte) and can be from 0 to 512 hexadecimal digits in length. When processing an output argument/variable with an odd number of hex digits the half byte after the last hex digit is cleared. For example, if an output variable is 5 hex digits, then when processing of the exec is completed, 5 hex digits corresponding to the vairable will be stored into the AXREXX invoker's storage followed by 4 bits set to zero.
5. AXRARGTYPEBITSTRING - The input is treated as a bit string. The length is specified in bits (8 per byte) and can be from 0 to 32. When processing an output argument/variable whose length is not a multiple of 8 bits, the bits following the bit string up to the next byte will be cleared to zeros. For example, if an output variable is 9 bits then when processing of the exec is completed, 9 bits of the AXREXX invoker's storage will be set to the variable followed by 7 bits of zeros.

If AXREXX encounters an error while attempting to marshal the invoker's input into a REXX argument or variable, System REXX will indicate in AXRARGLstEntryInError the number of the argument or variable that caused the error. AXREXX will return a specific reason code indicating the problem with the argument or variable and abort the request.

In addition to any input argument or variables that the AXREXX invoker may provide, System REXX sets the following variables:

1. AXREQTOKEN - Contains a 16 byte value which uniquely identifies the AXREXX invocation.
2. AXRINDD - If the REXXINDSN keyword is specified, this variable will contain the name of the DD used for allocating the input data set; otherwise it is not set.
3. AXROUTDD - If the REXXOUTDSN keyword is specified, this variable will contain the name of the DD used for allocating the output data set; otherwise it is not set.

If the exec successfully completes (no run time errors) and the AXREXX invocation specifies SYNC=YES, System REXX will attempt to obtain the final values of any output arguments or variables (those that have indicated AXRARGOUTPUT), convert them into the specified data type and insert their converted values into the AXREXX invoker's buffers specified by AXRARGADDR/AXRARGALET. The lengths of output arguments and variables are inserted into AXRARGOUTLENGTH. If there is any failure with attempting to process a single output argument or variable, System REXX will abort and not attempt to retrieve subsequent arguments or variables. Since output arguments are retrieved prior to output variables, if System REXX fails to process an output argument, no subsequent output arguments are processed and no output variables are processed.

In addition to output arguments and variables, System REXX also returns the return code from the exec in the AXRDIAGEXECRETCODE area in the REXXDIAG parameter (see AXRZARG for the mapping macro). The return code is returned as a 31 bit signed binary value. If it cannot be converted into such a value, or if the exec does not return a return code, then AXRDIAGNOEXECRETCODE will be set on.

Input/output files

AXREXX allows TSO=NO invokers to pass an input data set via the REXXINDSN parameter and both TSO=NO and TSO=YES invokers to specify an output data set via the REXXOUTDSN parameter. The input data set is used by REXX functions that require input from a user such as PARSE PULL or could be read directly via EXECIO, using the DDNAME specified by the REXX variable AXRINDD. If an output data set is specified, any SAY or TRACE output from the exec is directed there. Data may also be written to the output data set via EXECIO using the DDNAME specified by the REXX variable AXROUTDD. Any error message that the REXX interpreter issues will also be directed to the output data set. If no output data set is supplied, SAY, TRACE, and REXX messages will be directed to the console specified by the CONSNAME keyword as part of a multi-line WTO AXR0500I. The AXREXX user should be careful not to flood the system with messages and be careful when using REXX Tracing when the output is directed to a console. If CONSNAME and REXXOUTDSN are both not specified, the output is lost.

Both the input and output data sets may be sequential or partitioned. In the case of the latter, the REXXINMEMNAME or REXXOUTMEMNAME keywords must be specified. If the output data set does not exist, System REXX will create a sequential or partiitoned data set consisting of 3 primary blocks, 3 secondary blocks and 1 directory block (if it is a PDS) where each block is 27920 bytes. The data set will be kept when the exec completes and excess space will be released. The data set specified by RexxInDsn will be allocated by System REXX with DISP=SHR, while the data set specified by RexxOutDsn will be allocated with DISP=OLD (or NEW if it does not exist).

If System REXX detects that the output data set runs out of space, the exec will be terminated and a return code of 12 will be returned to the AXREXX invoker. If there is no data for the PARSE PULL instruction in the input data set, the null string will be returned.

Functions

Five functions are provided:

- AXRWTO
- AXRMLWTO
- AXRCMD
- AXRWAIT
- AXRINFO

AXRWTO is a function that issues the WTO macro containing the specified text, using the CONSNAME and CART provided on the AXREXX invocation. If CONSDATA is not specified the message text is sent to the system log. There is one required argument for AXRWTO:

- Message text - Containing the message text to be issued via the WTO macro. The text must be from 1 - 126 characters in length.

The return codes from AXRWTO are displayed as follows:

- 0 - AXRWTO was successful.
- 4 - Input message text was too long. WTO is truncated.
- 8 - Input message text was a 0 length string. WTO is not issued.

- 12 - Too many arguments were passed. WTO is not issued.
- 16 - The WTO macro returned a bad code. The REXX variable AXRDIAG contains the return code from WTO (in hexadecimal).
- 20 - Required argument was omitted. WTO is not issued.

The following are examples for AXRWTO:

```
WTOResult=AXRWTO('Hello from a REXX Exec');
CALL AXRWTO 'Hello from  a REXX Exec';     /* The return code from
                      AXRWTO is set in the variable "Result"   */
```

AXRMLWTO is similar to AXRWTO, except instead of issuing a single line WTO, a multi-line WTO is issued. There are three required arguments for AXRMLWTO:

1. Message Text - containing the text of the message and the length of which depends on the line type.
2. Name of connect id variable - for the first invocation of the multi-line WTO, the variable should be set to FIRSTLINE. After the first call, the variable will contain the connect id and should not be altered by the application until the multiline WTO is completed. This argument must appear in quotation marks.
3. Line type - there are 5 different line types. For details, see the WTO - Write to Operator macro in *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.
   - C - Control Line. The message text must be from 1-35 characters. If used, this must be the first line of the multi-line.
   - L - Label line. The message text must be from 1-71 characters.
   - D - Data line. The message text must be from 1-71 characters.
   - DE - Data end line. The message text must be from 1-71 characters.
   - E - End line. The message text must be omitted.

AXRMLWTO has the following return codes:

- 0 - AXRMLWTO was successful.
- 4 - Message text was too long. Message is truncated.
- 8 - Text length was 0. WTO not issued.
- 12 - too many arguments were passed. WTO not issued.
- 16 WTO macro returned a bad code. The REXX variable AXRDIAG contains the return code from WTO in hexadecimal.
- 20 - A required argument was omitted. WTO is not issued.
- 24 - Invalid line type. WTO is not issued.
- 28 - Invalid connect id. WTO is not issued.
- 32 - Unable to set connect id variable. The variable name is bad. WTO is issued.

The following are examples of the invocation of AXRMLWTO:

```
ConnectId = 'firstline';
MyResult=AXRMLWTO('Line 1 from a REXX Exec', 'Connectid','c');  /* Issue
                             the 1st line of the multiline */

IF MyResult^=0 THEN
  EXIT 1;
MyResult=AXRMLWTO('Line 2 from a REXX Exec', 'Connectid','d'); /* Issue
                             the 2nd line of the multiline    */
IF MyResult ^=0 THEN
  EXIT 2;
CALL AXRMLWTO ,'ConnectId','e';  /* End the multiline.  In this case,
                              return code from AXRMLWTO is set in the
                              variable "result"  */
```

AXRCMD is used to issue a system command from within the exec and obtain one or more command responses. The arguments that can be specified are as follows:

- Command text - The system command to be invoked. This is an optional argument. If it is omitted, no command will be issued, but a response from the last command issuance will be returned if one exists.
- Msgstem - The stem of a list of variables into which AXRCMD places the command response message text. This is an optional argument. If it is omitted, command text must be specified. To place the message text into compound variables which allows for indexing, msgstem should end with a period (for example, "messg."). AXRCMD places each line of the retrieved message into successive variables. For example, if the command response is a 3 line message, then messg.1 contains line 1, messg.2 contains line 2 and mssg.3 contains line 3. messg.0 will contain the number of lines. If msgtem does not end with a period, the variable names are appended with consecutive numbers. For example, suppose you specify msgstem as "conmsg" (without a period). If AXRCMD retrieves a message that has two lines of message text, AXRCMD places the text in the variables consmsg1 and consmsg2. The variable consmsg0 contains the number of lines in the message text, which is 2.
- Time - The amount of time in seconds that AXRCMD should wait for a command response. This is an optional argument. If it is omitted, AXRCMD will not wait before attempting to determine whether a command response was returned. A value of 0 - 21474535 seconds may be specified.

Return codes are as follows:

- 0 - The system command was issued and the command response received if requested. The REXX variable AXRDIAG contains the return code and ASID (separated by a blank) from MGCRE which are only valid for the START command.
- 4 - Command was issued but no command response was received in the specified time limit.
- 8 - The input wait time contained an invalid value. AXRCMD is not processed.
- 12 - Too many arguments were passed. AXRCMD is not processed.
- 16 - The variable name to contain the output of the system command was not acceptable to REXX. If a command was specified, it is issued, but no command response is returned.
- 20 - A required argument was omitted. Either the Command text or Msgstem (or both) must be specified. This return code will also occur if both of these arguments are 0 in length. AXRCMD is not processed.
- 24 - STIMERM could not be issued to time the request, possibly because there were too many STIMERM requests already in existence. The command was issued if the command argument was specified, but the command response is not returned.
- 32 - The command text is too long, exceeding 126 characters. AXMCMD not processed.

When command text is specified, AXRCMD invokes the MGCRE macro to issue the command. Once the command is successfully issued, AXRCMD will retrieve the command response. Most commands will issue the command response as a multi-line WTO. However, there are some commands that return the response in a number of distinct WTOs. In order to obtain all the command responses returned in distinct WTOs, the AXRCMD command can be reissued without any specified command until there are no more responses to obtain. For example, CmdResult=AXRCMD(,OutputVar.,Time). A return code of 4 in CmdResult indicates that there are currently no more responses to be obtained, although the command may still be providing responses.

When the START command is invoked, the AXRDIAG variable will contain the return code from MGCRE in hexadecimal, followed by the ASID of the new address space (also in hexadecimal), separated by a blank.

Users of AXRCMD can initiate another REXX exec by either specifying the MODIFY AXR command or by using the System REXX command prefix, but should be careful not to recursively invoke the same exec using AXRCMD.

The following are some examples of the invocation of AXRCMD.

This example will issue a command which returns its response as a multi-line WTO and will display the output from it:

```
CmdResult=AXRCMD ('D U,,,480','OUTPUTVAR.',40); /*Issue the D U command */

DO I = 1 TO OUTPUTVAR.0                      /* Iterate for the number
                                         of lines returned */
  SAY OutputVar.I                           /* Say the line   */
END;
```

This example will issue a command that returns its response in separate WTOs and will display the output from them. This can also be used to issue and display output for commands that return their responses in a multi-line WTO.

Note: Variable OutputVar.0 will only contain the number of lines within a single WTO issuance.

```
      CmdResult =  AXRCMD('$DI',OutputVar.,40)
      IF CmdResult = 0 THEN
         DO WHILE CmdResult = 0
            IF OutputVar.0 > 1 THEN
               DO LineNum = 1 TO OutputVar.0
                  CALL AXRWTO OutputVar.LineNum
               END
            ELSE
               CALL AXRWTO OutputVar.1
            CmdResult=AXRCMD(,OutputVar.,40) /* Wait up to 40 sec for
                             a cmd response */
         END
      ELSE
         CALL AXRWTO 'command failed. RC=' CmdResult
```

Below is an example of the use of AXRCMD, but not waiting for the command response. Instead, retrieving the command response at a later time:

```
MyResult=AXRCMD('D  T');     /* Issue the D T command but don't wait for
                             a command response   */
     /* Do some other work */
MyResult=AXRCMD(, OUTPUTVAR.,10);    /* Obtain the command response from
                            the last command issued and wait 10 seconds
                            if it hasn't already arrived */
```

AXRWAIT is a function that provides the capability for a REXX exec to wait for a specified amount of time in seconds. A single parameter with a numeric value between 0 and 21474536 is required. If the input is not syntactically correct, AXRWAIT ends the exec with a syntax error; otherwise, it returns error codes. The following statement would be coded to pause an exec for 15 seconds:

```
X = AXRWAIT(15)
```

Possible syntax error messages are:

AXRWAIT
Missing argument

AXRWAIT
Extraneous argument

AXRWAIT
Non-numeric time value

Possible return codes are:

- 14 - Wait time was too large.
- 18 - STIMERM returned a bad code. The variable AXRDIAG contains the code that STIMERM returned.

AXRINFO method returns information about the environment under which the exec is running. The following are valid options:

SUBSYSTEM
Returns the name of the Subsystem associated with the home address space the exec is running in.

AXREXXINVOKERSHOMESTOKEN
Returns the STOKEN associated with the home address space in which AXREXX was invoked to initiate the currently running exec.

REXXENVTYPE
Returns information about the environment that the exec is running under. The possibilities are:

- TSO=YES,TMP=NO
- TSO=YES,TMP=YES
- TSP=NO

The following is an example for the parameter option SUBSYSTEM:

```
X = AXRINFO('SUBSYSTEM')
```

Possible syntax error messages are:

AXRINFO
Missing argument

AXRINFO
Extraneous argument

AXRINFO
Unrecognized argument

Time limits and canceling a request

The AXREXX invoker can limit the amount of time that an exec can run by using the TIMELIMIT/TIMEINT keywords. When the time limit is reached, System REXX will invoke Halt Interpretation (HI) in the REXX environment where the exec is running. However, note that compiled execs must be compiled with the %TESTHALT option for the HI to work.

If the exec still does not complete after waiting for some time, the task running the exec will be detached. Invokers who specify a time limit should realize that time out is an error condition and that for SYNC=YES invokers, the final values of output arguments and variables will not be returned to the AXREXX invoker.

AXREXX supports an interface to CANCEL an exec. SYNC=NO AXREXX invokers can obtain the Request Token via the OREQTOKEN parameter for later input to AXREXX CANCEL. Cancel is processed as if the exec timed out.

Error handling

If a syntax or run time error is encountered during the execution of the exec and the exec does not trap on it, System REXX will return to the AXREXX invoker with a return code of X'8', reason code X'xxxx0828'. The AXREXX invoker should look at the data set specified on the REXXOUTDSN parameter for any error messages. If the invoker specified CONSDATA, but not REXXOUTDSN, the error messages will be directed to the console specified by the CONSNAME parameter. The message id corresponding to syntax error message IRXnnnnI is inserted into AXRDIAG1 of the REXXDIAG parameter, the line number is inserted into AXRDIAG2, and the message IDs of the last 2 IRX or IKJ messages that were issued are inserted into AXRDIAG3 and AXRDIAG4. When the exec completes successfully and AXREXX returns a code of 0 then AXRDIAG1, AXRDIAG2, AXRDIAG3 and AXRDIAG4 will contain the message ids of the last 4 IRX or IKJ messages issued while the exec was running. The format of these message ids is packed decimal with the sign bits shifted out. A value of 1 in the high order byte distinguishes an IKJ message from an IRX message. The REXXDIAG parameter of the AXREXX macro is mapped by the AXRDIAG in AXRZARL.

System REXX execs should release any obtained resources prior to exiting. In the event the exec is aborted due to a syntax error, time out or CANCEL, the task under which the exec is running will be terminated; thus any task related resources that were obtained by the exec will be freed. In the TSO=YES environment, any data sets that were left allocated by the exec are unallocated. Any other address space or system resources are not freed by System REXX.

Examples

The following is an example of an assembler program using AXREXX to invoke a REXX exec to parse the output of DISPLAY JOBS to obtain the address of the ASTE of *MASTER*.

```
GETASTE  CSECT ,
GETASTE  AMODE 31
GETASTE  RMODE 31
*
*  TITLE: GetAste
*
*  Function:  Obtain the address of MASTER's aste by invoking
*     an exec to parse the output of DISPLAY JOBS,*MASTER*.
*     The following exec takes a jobname as an input argument
*     and sets the variable OutAste@.
*
*
* NUMERIC DIGITS 25
* ARG InJobname
* MyCmd = 'D JOBS,' || Strip(InJobname);
* Result =  AXRCMD(MyCmd,OutputVar.,10);
* IF Result = 0 THEN
*   DO;
*     OutAste@ = ' '
*     DO LineNum = 1 TO OutputVar.0 WHILE(OutASTE@=' ');
*       PARSE var OutputVar.LineNum 'ASTE=' OutAste@
*     END;
*     IF OutAste@ = ' ' THEN
*       DO;
*         MyRetcode = 8;
*         OutAste@ = 0;
*       END;
*     ELSE
*        MyRetcode = 0;
*   END;
* ELSE
*   DO;
*     MyRetcode = 12;
*     OutAste@ = 0;
*   END;
* EXIT MyRetcode;
*
*
*
************************************************************
         BAKR  14,0
         USING GETASTE,12
         LR    12,15
         MODID BR=YES
         XC  MyArgLst,MyArgLst    Clear the ArgLst header
         XC  MyVarLst,MyVarLst    Clear the VarLst header
         XC  MyArgEn1,MyArgEn1    Clear the Arg entry
         XC  MyVarEn1,MyVarEn1    Clear the Var entry
         LA  2,MyArgLst
         USING AxrArgLst,2
         MVC AxrArgLstId,MyAxrArgLstAcro
         LA  5,AxrArgLstCurVer
         ST  5,AxrArgLstVer       Initialize the version
         LA  5,kNumArgs           Obtain the number of arguments
         STH  5,AxrArgLstNumber   Store the number of arguments
         DROP 2
         USING AxrArgEntry,2
         LA  2,MyArgEn1          Addressability to first arg entry
         LA  5,kMaster
         ST  5,AXRARGADDRLOW      Store address of jobname (*master*)
         OI  AXRARGLSTINPUTFLGS1,AXRARGINPUT Indicate input arg
         LA  5,L'kMaster          Obtain length of arg
         ST  5,AXRARGLENGTH       Store length of arg in entry
         MVI AxrArgType,AxrArgTypeChar  Store type of arg
         DROP 2
         LA  2,MyVarLst
         USING AxrArgLst,2
         MVC AxrArgLstId,MyAxrVarLstAcro
         LA  5,AxrArgLstCurVer
         ST  5,AxrArgLstVer       Initialize the version
         LA  5,kNumVars           Obtain the number of variables
         STH  5,AxrArgLstNumber   Store the number of variables
         DROP 2
         USING AxrArgEntry,2
         LA  2,MyVarEn1           Addressability to 1st var entry
         LA  5,OutAste@
         ST  5,AXRARGADDRLOW      Store output argument
         LA  5,OutArgName
         ST  5,AXRARGNameADDRLOW Store address of name of output var
         MVI AxrArgNameLength,L'OutArgName
         OI  AXRARGLSTINPUTFLGS1,AXRARGOutput   Indicate output var
         MVI AxrArgType,AxrArgTypeHexString     Indicate hex string
         LA  5,L'OutAste@         Obtain length (in bytes)
         SLL 5,1       Mult by 2 - length is in hex digits (not bytes)
         ST  5,AxrArgLength       Store length in var entry
         DROP 2
         AXREXX REQUEST=EXECUTE,NAME=kEXECNAME,REXXARGS=MyArgLst,      *
               REXXVARS=MyVarLst,REXXDIAG=MyAxrDiag
         LTR   15,15
         JNZ   FailLabel
         USING AxrDiag,2
         LA    2,MyAxrDiag
         TM    AxrDiagFlgs1,AxrDiagNoExecRetCode
         JNZ   FailLabel
         L     15,AxrDiagExecRetCode
         LTR   15,15
         JNZ   FailLabel
*        Everything looks good.  Process OutAste@ here
FailLabel DS   0H
* Perform error checking
* OutAste@ should contain Master's ASTE address
         PR
kNumArgs EQU   1
kNumVars EQU   1
         DS    0D
MyAxrArgLstAcro DC AL4(AxrArgLstAcro)
MyAxrVarLstAcro DC AL4(AxrVarLstAcro)
kExecName DC CL8'GETASTE '
kMaster  DC CL8'*MASTER*'
OUTArgName DC CL8'OUTASTE@'
MyArgLst DS CL(AXRARGLST_LEN)
MyArgEn1 DS CL(AXRARGENTRY_LEN)
MyVarLst DS CL(AXRARGLST_LEN)
MyVarEn1 DS CL(AXRARGENTRY_LEN)
MyAxrDiag DS CL(AXRDIAG_LEN)
OutAste@ DS A
         AXRZARG DSECT=YES,AXRARGLST=YES,AXRARGENTRY=YES,AXRDIAG=YES
         END
```

The following shows an example of a program which sets up an extended MCS console to receive messages passed back from the exec. In this case, the exec sends back a portion of the output of a system command.

```
MCSOPER CSECT ,
MCSOPER AMODE 31
MCSOPER RMODE 31
*
*  TITLE: MCSOPER
*
*  Function:  Set up an EMCS console.  Invoke AXREXX, passing
*     it the console name and pass 2 arguments to the exec:
*     the command text to invoke and the number of lines of the
*     output of the command return to the invoker's EMCS console.
*
*
*  /* REXX */
* ARG InCmdText,InNumLines
* AxrCmdRc = AXRCMD(InCmdText,MsgStem.,4);
* IF AxrCmdRc = 0 THEN
*   DO;
*     ConnectId = 'FirstLine'
*     CALL AXRMLWTO 'Start of cmd output','ConnectId','C'
*     DO I = 1 TO InNumLines
*       CALL AXRMLWTO Msg.i,'ConnectId','D'
*     END;
*     CALL AXRMLWTO  ,'ConnectId','E'     /* End line */
*   END;
* EXIT AxrCmdRc
*
*
*
*********************************************************************
         BAKR  14,0
         USING MCSOPER,12
         LAE   12,0(12,0)
         LR    12,15
         MODID BR=YES
         XC  MyArgLst,MyArgLst
         XC  MyArgEn1,MyArgEn1
         XC  MyArgEn2,MyArgEn2
         LA  2,MyArgLst
         USING AxrArgLst,2
         MVC AxrArgLstId,kAxrArgLstAcro
         LA  5,AxrArgLstCurVer
         ST  5,AxrArgLstVer       Initialize the version
         L    5,kNumArgs           Obtain the number of arguments
         STH  5,AxrArgLstNumber   Store the number of arguments
         DROP 2
         USING AxrArgEntry,2
         LA  2,MyArgEn1          Addressability to first arg entry
         LA  5,kInCmdText
         ST  5,AXRARGADDRLOW      Store address of jobname (*master*)
         OI  AXRARGLSTINPUTFLGS1,AXRARGINPUT Indicate input arg
         LA  5,L'kINCmdText         Obtain length of arg
         ST  5,AXRARGLENGTH       Store length of arg in entry
         MVI AxrArgType,AxrArgTypeChar  Store type of arg
         DROP 2
         USING AxrArgEntry,2
         LA  2,MyArgEn2           Addressability to 2nd arg entry
         LA  5,InNumLines
         ST  5,AXRARGADDRLOW      Store number lines
         OI  AXRARGLSTINPUTFLGS1,AXRARGInput    Indicate input arg
         MVI AxrArgType,AxrArgTypeUnsigned      Indicate hex string
         LA  5,L'InNumLines       Obtain length (in bytes)
         ST  5,AxrArgLength       Store length in arg entry
         DROP 2
         MODESET MODE=SUP
         MCSOPER   REQUEST=ACTIVATE,CONSID=MyConsid,TERMNAME=kTERM,    *
               NAME=kConsname,MCSCSA=MyMcsCsa@,MCSCSAA=MyMcsCsaAlet,   *
               MsgEcb=MyMsgEcb
         AXREXX REQUEST=EXECUTE,NAME=kEXECNAME,REXXARGS=MyArgLst,      *
               REXXDIAG=MyAxrDiag,CONSDATA=YES,CONSNAME=kConsName,     *
               CART=kCart
         SAC 512
         SYSSTATE ASCENV=AR
         MCSOPMSG  REQUEST=GETMSG,CONSID=MyConsid
*        Process the MDB pointed to Reg1/AR1
*        The MDB will contain the messages issued by the exec
         SYSSTATE ASCENV=P
         SAC 0
         MCSOPER   REQUEST=DEACTIVATE,NAME=kConsName
         MODESET MODE=PROB
         PR
kExecName DC CL8'MCSOPER '
kConsname DC CL8'TEST1   '
kTerm    DC  CL8'TEST1'
kCart    DC F'1'
kNumArgs DC F'2'
kAxrArgLstAcro DC AL4(AxrArgLstAcro)
kInCmdText DC CL12'D JOBS,ALL  '
InNumLines DC F'10'
MyConsid DS F
MyMsgEcb DS F
MyMcsCsa@ DS F
MyMcsCsaAlet DS F
MyAxrDiag DS CL(AXRDIAG_LEN)
MyArgLst DS CL(AXRARGLST_LEN)
MyArgEn1 DS CL(AXRARGENTRY_LEN)
MyArgEn2 DS CL(AXRARGENTRY_LEN)
         AXRZARG DSECT=YES,AXRDIAG=YES,AXRARGLST=YES,AXRARGENTRY=YES
         END
```

Chapter 32. z/OS FBA services

z/OS FBA services provides the ability to manage and perform I/O on z/OS systems, exploiting the z/OS Distributed Data Backup (zDDB) multi-platform access feature on DS8700 and subsequent model type devices. The new application programming interface (API) provides functions to allocate, read from, write to and deallocate z/OS FBA devices.

z/OS FBA devices are defined using the Hardware Configuration Dialog (HCD) or Hardware Configuration Manager (HCM). z/OS FBA devices must be online in order to be used by z/OS FBA Services.

z/OS FBA devices can be used in many ways. For example, z/OS FBA devices can be used as a data transfer device between a z/OS system and a distributed system. It can also be used as a shared data device between z/OS and other systems. The z/OS FBA service encapsulates the management of z/OS FBA devices and provides capability for I/O channel program creation and execution from z/OS systems.

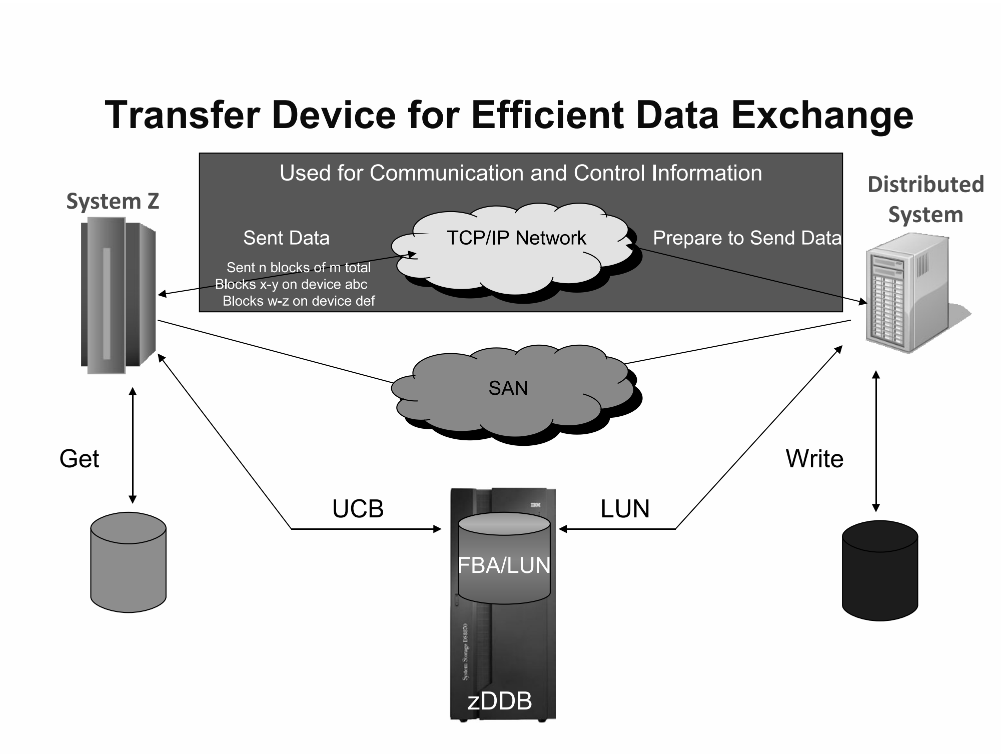


Figure 86. z/OS FBA for data transfer

z/OS Distributed Data Backup (zDDB)

The z/OS I/O subsystem communicates with the disk controller by sending out Channel Command Words (CCWs). These disk devices are setup for Extended Count Key Data (ECKD™) format.

zDDB is an optional licensed feature introduced in 2010 for DS8700 storage controllers (and subsequent models) that allows devices on hosts attached through FICON® interfaces to access data on fixed block (FB) volumes. Typically, FBA disk devices are connected to Linux, UNIX®, and Windows® operating systems.

With zDDB, two views of the disk devices are presented, one for z/OS and one for the distributed system.

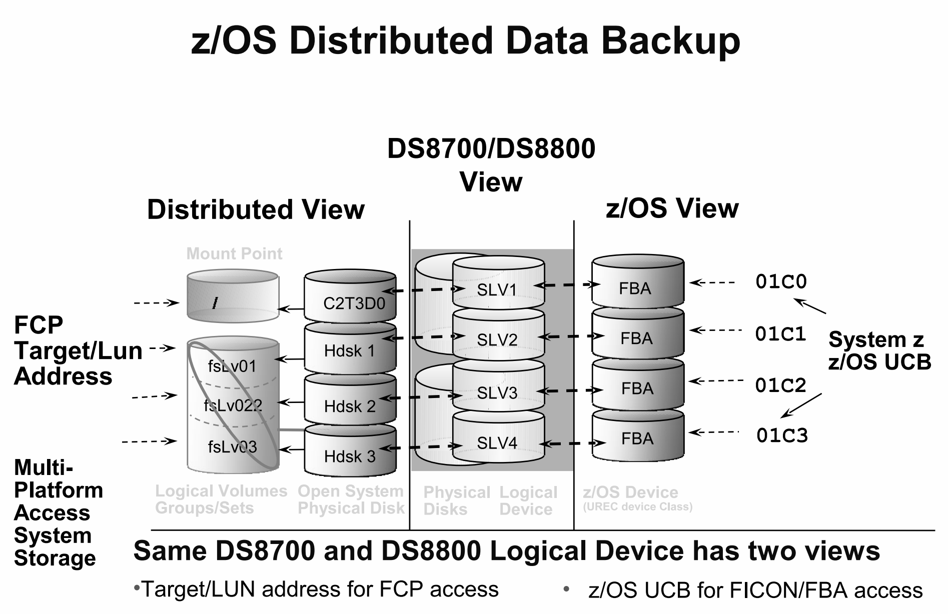

z/OS Distributed Data Backup
DS8700/DS8800 View
Distributed View
z/OS View
Mount Point
/
C2T3D0
Hdsk 1
Hdsk 2
Hdsk 3
fsLv01
fsLv022
fsLv03
SLV1
SLV2
SLV3
SLV4
FBA
FBA
FBA
FBA
01C0
01C1
01C2
01C3
System z z/OS UCB
FCP Target/Lun Address
Multi-Platform Access System Storage
Logical Volumes Groups/Sets
Open System Physical Disk
Physical Disks
Logical Device
z/OS Device (UREC device Class)
Same DS8700 and DS8800 Logical Device has two views
•Target/LUN address for FCP access
• z/OS UCB for FICON/FBA access

Figure 87. z/OS Distributed Data Backup

For more information about the zDDB feature of the DS8700 storage controller, see the DS8K information center.

z/OS FBA devices

In order to use z/OS FBA devices, the following must occur:

- The zDDB licensed feature must be installed on the storage controller where z/OS FBA devices will be used.
- z/OS FBA devices must be defined for the active configuration.
- z/OS FBA devices must be online.

z/OS FBA devices are implemented as a new device or control unit type in the Unit Record (UR) class of devices. To configure these devices in HCD, first create a new control unit with TYPE = 2107-FBA. Once the control unit is properly defined, create new devices of TYPE=FBA attached to this control unit. Given that these devices are defined as Unit Record devices, they cannot be shared among users on a single system.

z/OS FBA devices can be designated as "online at IPL" (the default) or can be explicitly varied online using the VARY device command. *Devices must be online before being used by z/OS FBA services.*

z/OS FBA devices may be used by various software products. You should consult with software product documentation to understand requirements for z/OS FBA devices.

Note that Unit Control Blocks (UCBs) for z/OS FBA devices have UCBTBYT3=UCB3UREC and UCBTBYT4 = X'60'.

z/OS FBA devices should only be shared by systems within a single sysplex. The IOSFBA service uses systems ENQs to serialize the allocation process. Sharing z/OS FBA devices with systems in different sysplexes may cause allocation issues.

Controlling access to z/OS FBA devices

Access from z/OS systems to z/OS FBA devices may be controlled using a standard device security profile if desired. See *Security Server Administrator's Guide* for more details. An example using RACF to control access to z/OS FBA device 2000 is:

RDEFINE DEVICE (*.UR.FBA.2000) UACC(NONE)

Remember that the device's class must be on the RACLIST and refreshed for this checking to be in effect. Once this resource is defined, the PERMIT command can be used to grant access to users.

Access to LPARs may also be controlled using device candidate lists in HCD. For more information, see *z/OS HCD User's Guide*.

Access to z/OS FBA devices from distributed systems may also be controlled. Storage network administrators can use fabric zoning to control what systems can access the devices. Storage administrators can also use LUN masking.

When z/OS FBA devices are to be used by both z/OS and distributed systems, the software product using the devices can provide security controls dynamically using the SCSI Persistent Reserve command. Using Persistent Reserve, the software product's distributed client can isolate access to the disk from other distributed systems without affecting z/OS's access to the device.

The combination of using Persistent Reserve, along with security product controls from z/OS, can create a secure environment for storing data on z/OS FBA devices.

z/OS FBA services

z/OS FBA services are supported by the IOSFBA macro. IOSFBA is an authorized, supervisor state service that provides a means for callers to manage (allocate and unallocated) and perform I/O to (read and write) allocated FBA devices. IOSFBA is used in conjunction with the new IOSDFBA and IOSDIOST mapping macros. IOSFBA must be invoked in task mode and enabled for I/O interrupts; it can be called in 31-bit or 64-bit addressing mode.

The IOSFBA API provides the following functions:

- QUERY
- ALLOCATE

- READ
- WRITE
- ERASE
- CLEANUP
- UNALLOCATE

Because some of the areas allocated by the IOSFBA service are task related, all IOSFBA service calls must be made from the same address space and task when using the same set of devices to ensure that access to the appropriate control blocks is available.

Querying and allocating FBA devices

Querying FBA devices

The QUERY function lets the caller gather physical and self-describing information on the specified devices and an indication if the FBA device is allocated. The caller must provide a list of devices to query.

The information returned includes the starting block and number of blocks available for each device, the physical record size (size of each block), and the I/O Node Element Descriptor (I/O NED).

Using IOSFBA QUERY to gather information on z/OS FBA devices

First, a program must know what devices are defined for its use. Typically, this would be done using either parameters or an initialization file. The program could also perform a UCBSCAN with DEVCLASS=UREC looking for devices with UCBTBYT4=X’60’. Using an initialization file or parameters is the preferred method because many software products may be using z/OS FBA devices on a single system.

Once devices to be used are known, the program would build a list of devices (mapped by the FBADL DSECT in the IOSDFBA macro) and invoke the IOSFBA QUERY function.

If successful, IOSFBA QUERY returns a device descriptor list (mapped by the FBADDL DSECT in the IOSDFBA macro) and one or more FBADDE structures describing the devices.

The storage containing the FBADDL structure must be freed by the caller when the information is no longer required. The length field in the FBADDL header is the total length of storage, including storage for the FBADDE structures.

For input and output to the IOSFBA QUERY service, see Figure 88 on page 821.

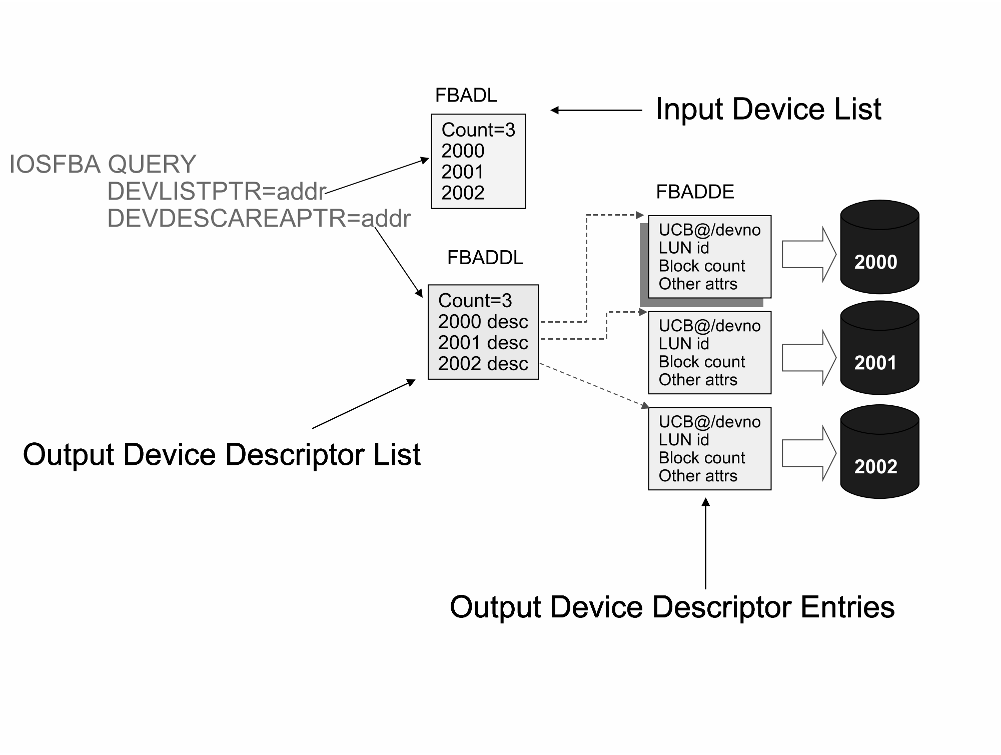


Figure 88. Visual representation of IOSFBA query

Allocating FBA devices

An FBA device must be online and allocated prior to it being used to read, write, or erase data from the device.

As with the QUERY function, ALLOCATE requires a list of devices that are to be allocated. The function dynamically allocates (SVC99) the device and provide the same information that the QUERY function provides.

The information returned after the IOSFBA ALLOCATE service is issued contains the same information returned from the IOSFBA QUERY service (the starting block and number of blocks available for each device, the physical record size (size of each block), and the I/O Node Element Descriptor (I/O NED)), plus additional information specific to the IOSFBA ALLOCATE service. This information is used as input for the READ, WRITE and UNALLOCATE functions.

For input and output to the IOSFBA ALLOCATE service, see Figure 89 on page 822.

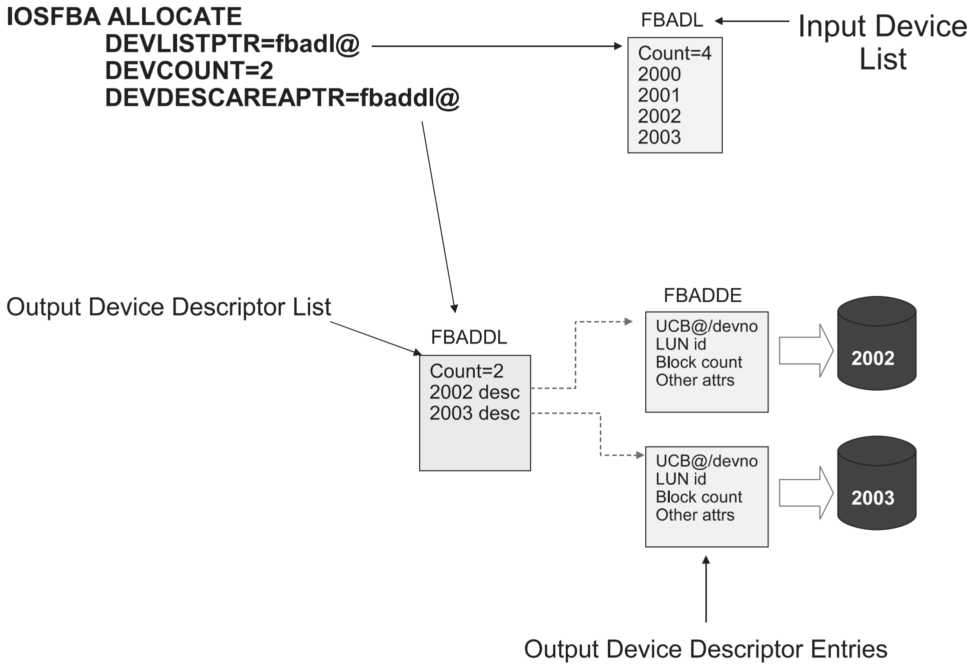


Figure 89. Visual representation of IOSFBA allocation

How the device is to be used dictates how it should be allocated. The ALLOCATE function provides the following access types for allocating a z/OS FBA device:

- Specifying ACCESS=SINGLE limits the usage of the device to a single z/OS system within the SYSPLEX.
- Specifying ACCESS=READ or ACCESS=WRITE allows the usage of the device to two z/OS systems within the SYSPLEX (one as READ and the other as WRITE).
- Specifying ACCESS=ANY provides the caller control of how the device is serialized on the z/OS systems within the SYSPLEX.

For all access types, distributed systems can still share the device; that is, the distributed system can read from or write to the device.

Note that with ACCESS=SINGLE, ACCESS=READ, or ACCESS=WRITE, a small amount of metadata is written to the device. Ensure that important data that may reside on the disk is not erased by the IOSFBA service. If data exists on the z/OS FBA disk, using ACCESS=ANY ensures that the IOSFBA does not alter that pre-existing data.

When selecting the method of allocation by the calling programs, the pool of devices used for ACCESS=ANY should not be mixed with the pool of devices used for ACCESS=SINGLE, ACCESS=READ, or ACCESS=WRITE allocations. This avoids duplicate allocations from different systems within a sysplex because ACCESS=ANY does not use metadata to control device allocations

Using the IOSFBA ALLOCATE function to allocate z/OS FBA devices

A program must know what devices are defined for its use. Typically, this is done using either parameters or an initialization file. The program can also perform a

UCBSCAN with DEVCLASS=UREC looking for devices with UCBTBYT4=X’60’. Using an initialization file or parameters is the preferred method because many software products may be using z/OS FBA devices on a single system.

Once devices to be used are known, the program builds a list of devices (mapped by the FBADL DSECT in the IOSDFBA macro) and invokes the IOSFBA ALLOCATE function.

If successful, IOSFBA ALLOCATE returns a device descriptor list (mapped by the FBADDL DSECT in the IOSDFBA macro) and one or more FBADDE structures describing the devices that were allocated. The FBADDL is required input for the IOSFBA UNALLOCATE service.

The storage containing the FBADDL structure must eventually be freed by the caller when the information is no longer required. This should be done by the calling program after calling the IOSFBA UNALLOCATE service. The length field in the FBADDL header is the total length of storage, including storage for the FBADDE structures.

Identifying FBA devices with distributed SCSI logical unit numbers (LUNs) and other z/OS systems

Part of the information returned by both IOSFBA QUERY and IOSFBA ALLOCATE is an I/O Node Element Descriptor (NED). The I/O NED is 32 bytes long. It uniquely identifies the FBA device and contains:

- The device manufacturer
- The plant where the device was built
- The model number
- The sequence number
- The logical subsystem
- The unit address

For distributed environments, this information can be obtained by issuing a standard SCSI inquiry command. Remember that the information returned in a z/OS environment is in EBCDIC format, while distributed environments return the information in ASCII format.

Reconciling the information from both sources helps the peer programs ensure that the correct device is being used.

Reconciling information from both sources to help the peer programs ensure that the correct device is being used

Input to QUERY and ALLOCATE

The QUERY and ALLOCATE functions require the caller to pass the address of an FBA device list, which is mapped by the FBADL (found in the IOSDFBA mapping macro). The FBADL consists of two structures; the first can be referred as the header (FBADL) and the second can be referred to as the device entry (FBADL_DEVICE_ENTRY). The FBADL header indicates the length of each entry, the total length (header and entries), and the number of device entries. For each device to be queried or allocated, a device entry must exist.

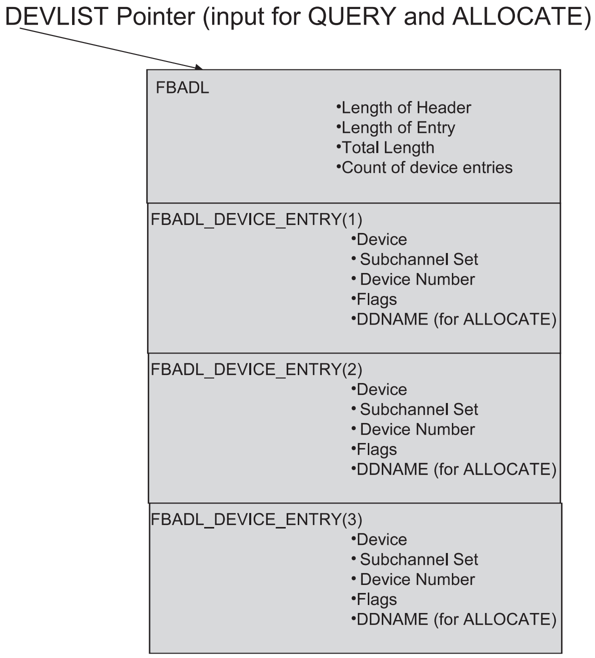


Figure 90. FBADL with three device entries

Output from QUERY and ALLOCATE

The output from either the IOSFBA QUERY or IOSFBA ALLOCATE request is an address of a storage area obtained by the IOSFBA service that maps the FBA device descriptor list (FBADDL) and the FBA device descriptor entry (FBADDE). The FBADDL and FBADDE provide detailed information about each FBA device that was either queried or allocated. This information is required input for READ, WRITE, and UNALLOCATE options. The caller or invoker is responsible for releasing the storage through either a STORAGE RELEASE invocation for callers in AMODE 31 or an IARST64 invocation for callers in AMODE 64.

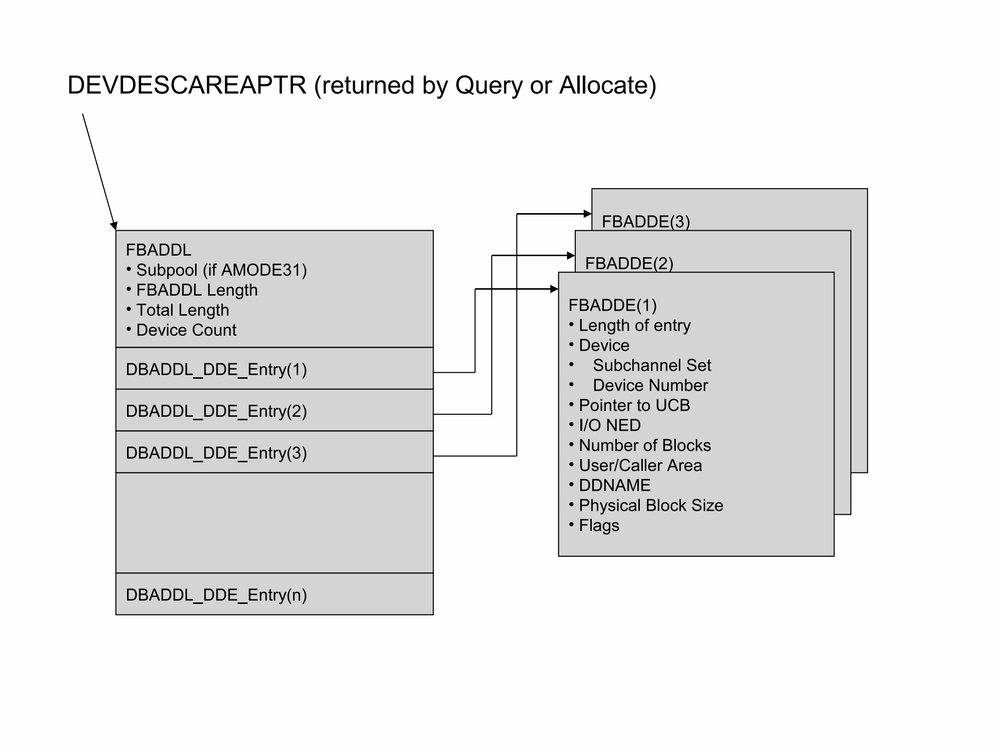


Figure 91. Mapping of output area from IOSFBA QUERY and IOSFBA ALLOCATE

The FBADDE structures is pointed to by the FBADIOE structures created to read or write to the z/OS FBA devices.

Reading from and writing to z/OS FBA devices

To read from or write to an z/OS FBA device or set of z/OS FBA devices, a z/OS FBA device I/O list (FBADIOL) entry for each device is created that contains, among other things, a device descriptor (returned by the ALLOCATE function), an I/O status block (mapped by the IOSDIOST), the storage buffers, and the blocks on the device that are to be read from or written to. The IOSFBA API will build the appropriate channel program and start the I/O. The read and write functions can be synchronous or asynchronous.

For asynchronous reading and writing, an ECB is passed to the IOSFBA service and control is returned to the invoking program once the I/O is started. The invoking program must wait, at some point, for the ECB to be posted. The ECB is posted once all I/Os are completed, either successfully or unsuccessfully, and the status of the I/O for the device is stored in the corresponding status block.

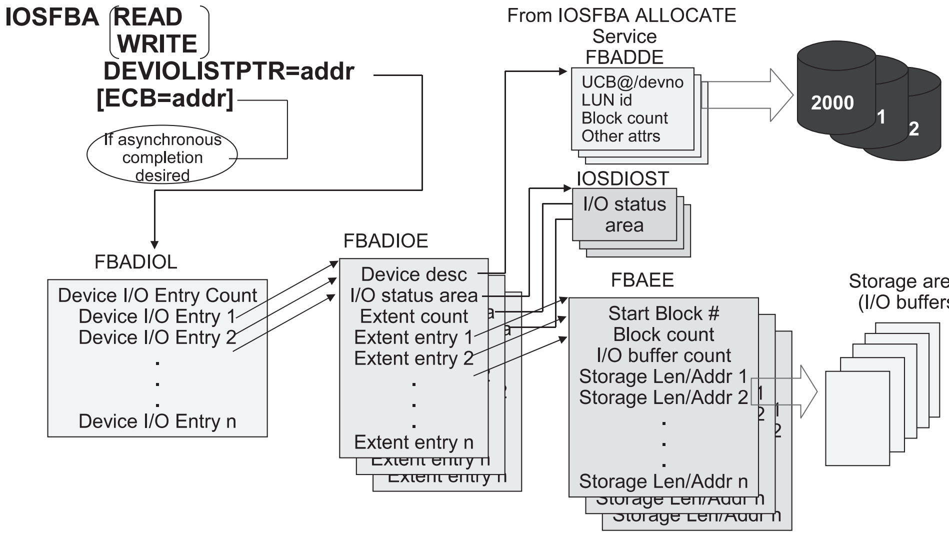


Figure 92. Visual representation of reading or writing to z/OS FBA devices

Input to IOSFBA READ and IOSFBA WRITE requests

When reading from or writing to an z/OS FBA device, a z/OS FBA device I/O list (FBADIOL) specifies the I/O to be performed. The FBADIOL contains a count of the devices that are involved with the READ or WRITE request and an address for each I/O entry represented by the FBADIOE. An FBADIOE is the representation of I/O to be performed to an FBA device (represented by the FBADDE). The FBADIOE contains pointers to extent entries (FBAEE) that indicate the starting location on the FBA device and length of data to be read or written. The data itself is represented as buffer entries (FBAEE_BUFENT) that contain the location and length of specific sections of data. For READ operations, it indicates the location and length of where data will be read from. For WRITE operations, it indicates the location and length of where data will be written to. These control blocks are found in the IOSDFBA mapping macro.

There are two general rules regarding the storage buffer entries:

- Each buffer entry must be a multiple of the physical block size. Partial block sizes cannot be read or written.
- The number of target blocks on the device must not be exceeded by the number of blocks represented by the storage buffer entries.

The service also provides the capability of determining the completion status of each of the I/Os (FBADIOEs) through the use of the status block address. For each FBADIOE, IBM recommends that a status block be allocated and the address stored in the FBADIOE. The status block is mapped by the IOSDIOST mapping macro. It contains the I/O completion code, reason code detailing the completion code value, device and subchannel status, and sense data (when device status contains a unit check).

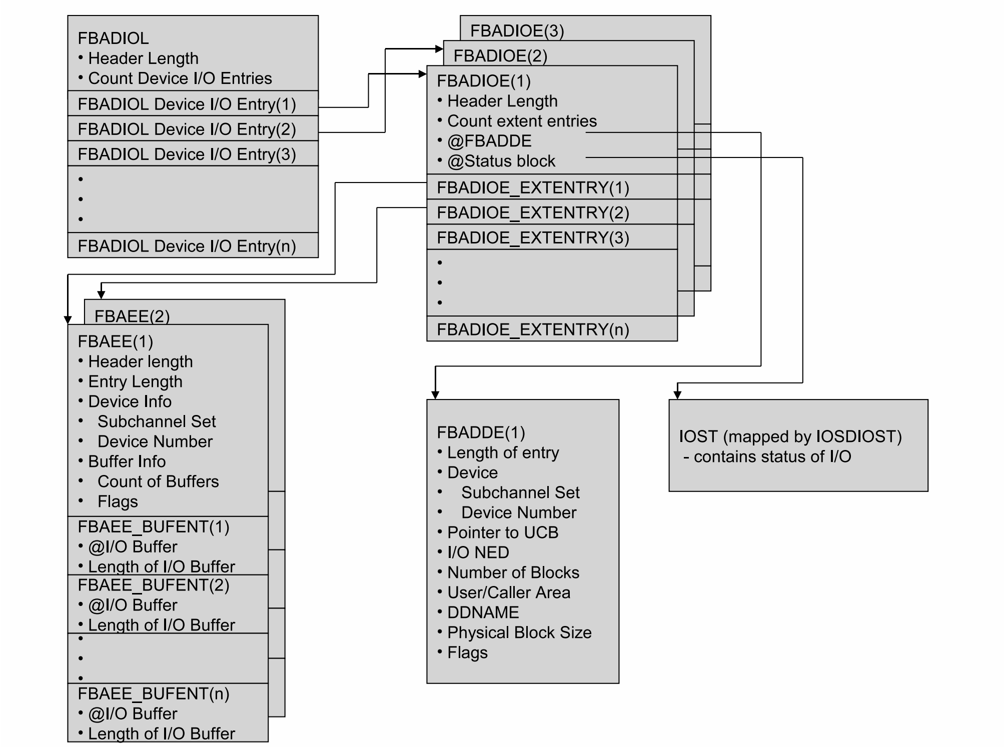


Figure 93. Controls blocks needed for FBA I/O

Erasing data on z/OS FBA devices

It is possible to use the z/OS FBA devices when using the devices as transfer devices. IOSFBA READ/WRITE remembers the ranges of blocks used on the z/OS FBA devices, allowing IOSFBA ERASE to construct a channel program to erase the data. This channel program erases all data from the lowest to highest contiguous blocks referred to in the IOSFBA READ/WRITE processing.

It may be more efficient for the calling program to zero the storage buffers used and invoke the IOSFBA WRITE service to write zeros to the blocks used.

Caution should be used by the calling program to ensure that data is not inadvertently erased.

Unallocating z/OS FBA devices

Once all processing of the z/OS FBA device, including reading, writing, and erasing, is completed, it must be unallocated. Specifying the UNALLOCATE function unallocates the specified z/OS FBA device as indicated by the FBADDL that was created and returned when the z/OS FBA device was allocated.

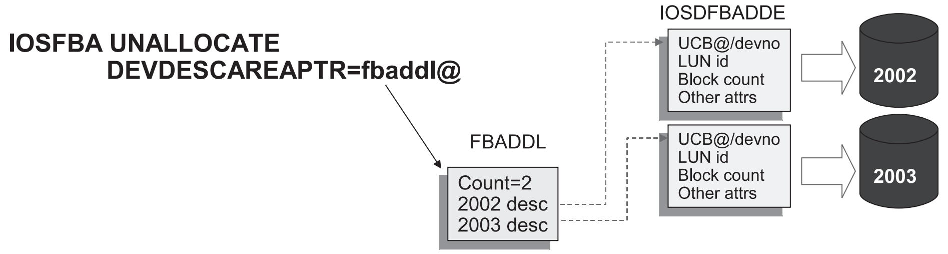


Figure 94. Visual representation of unallocating z/OS FBA devices

The input for the IOSFBA UNALLOCATE request is an address of a storage area returned by the IOSFBA ALLOCATE service. It maps the FBA Device Descriptor List (FBADDL) and the FBA Device Descriptor Entry (FBADDE). The FBADDL and FBADDE provide detailed information about each FBA device that was allocated. The caller or invoker is responsible for releasing the storage through either a STORAGE RELEASE invocation for callers in AMODE 31 or IARST64 invocation for callers in AMODE 64. The length field in the FBADDL header is the total length of storage, including storage for the FBADDE structures.

Improving performance when using IOSFBA services

It is possible to improve performance of the IOSFBA service by using the IOTOKEN keyword. The IOTOKEN keyword allows the caller to supply the address of a 32-byte area that will be used by the IOSFBA service to save information between IOSFBA service calls.

Performance may also be improved if a channel program can be reused. This is possible if all of the following conditions are true:

- The storage buffers and target blocks of the z/OS FBA devices are exactly the same.
- The address space where the program is located is non-swappable across all IOSFBA READ/WRITE requests.
- The storage buffers remain fixed pages across all IOSFBA READ/WRITE requests.

When this is the case, the REUSECP keyword can also be used. REUSECP in combination with the IOTOKEN keyword allows the IOSFBA READ/WRITE service to simply start the channel program without a channel program build.

More than one IOTOKEN can be used by calling programs if parallel IOSFBA invocations are used, if different channel programs must be built, or if different storage buffers or z/OS FBA device target blocks are used.

If IOTOKEN is used, the IOSFBA CLEANUP service must also be used to ensure that storage is properly released. The program must call IOSFBA CLEANUP for each IOTOKEN that was used.

Using more that one device for each IOSFBA READ/WRITE operation also helps to improve overall I/O throughput.

Providing a recovery or resource manager

A recovery or resource manager needs to be implemented by callers of the IOSFBA service. This resource manager is responsible for unallocating allocated devices and releasing storage when the application fails or abnormally terminates.

Devices that are allocated by an application remain allocated until a specific unallocate request is made. If the caller of the IOSFBA service fails and does not unallocate the devices, subsequent allocation attempts fail. This is especially true when ACCESS=SINGLE, ACCESS=READ, or ACCESS=WRITE was specified when initially allocating the device.

Appendix A. Using the unit verification service

The information in this topic describes using the unit verification service to obtain information from the eligible device table. IBM recommends that you use the EDTINFO macro instead; EDTINFO provides more services and is easier to use than the unit verification service.

EDTINFO **must** be used to obtain information on units that are defined as:
- Dynamic,
- Have 4-digit device addresses, or
- Are described by unit control blocks (UCBs) that reside above the 16-megabyte line.

The IEFEB4UV routine interface maybe used, only, to obtain information on units that are static, have 3-digit device addresses and are described as UCBs residing below the 16-megabyte line.

The unit verification service enables you to obtain information from the eligible device table (EDT) and to check your device specification against the information in the EDT. See *z/OS HCD Planning* for information on the EDT.

There are three versions of the unit verification service:
- IEFEB4UV, for problem programs or authorized callers.
- IEFGB4UV and IEFAB4UV, for authorized callers.

Note: IEFEB4UV is the preferred interface.

Functions

The unit verification service performs the following functions:
- Check groups
- Check units
- Return unit name
- Return unit control block (UCB) addresses
- Return group ID
- Indicate unit name is a look-up value
- Return look-up value
- Convert device type to look-up value
- Return attributes
- Check units with no validity bit
- Specify subpool for returned storage
- Return unit names for a device class

Check groups - Function code 0

This function determines whether the input device numbers make a valid allocation group. To be valid, the device grouping must include either all the device numbers being verified, or none of them. If this is not the case, the allocation group is split, and the input device numbers do not make up a valid allocation group.

Check units - Function code 1

This function determines whether the input device numbers correspond to the unit name in the EDT. In addition to a return code in register 15, it sets to one the high-order flag bit of any device numbers in the parameter list that are not valid.

Return unit name - Function code 2

This function returns the unit name associated with a look-up value provided as input. The unit name is the EBCDIC representation of the IBM generic device type (for example, 3390) or the esoteric group name (for example, TAPE) from the EDT.

A look-up value is an internal representation of the unit name, used as an index into the EDT. Because teleprocessing devices do not have generic device names, you cannot use this function to request information about teleprocessing devices.

Note: Do not use this function to determine whether a returned unit name is a generic CTC device or an esoteric group name that contains CTC devices. Instead, use the return attributes function (function code 8) for this purpose.

Return unit control block (UCB) addresses - Function code 3

This function returns the UCB pointer list associated with the unit name provided as input.

Return group ID - Function code 4

This function returns the allocation group ID corresponding to each UCB address specified in the input list.

Indicate unit name is a look-up value - Function code 5

The input to the check units and return UCB addresses functions can be specified as a four-byte internal representation of the unit name rather than as the unit name itself.

Return look-up value - Function code 6

This function returns the four-byte internal representation of the unit name that serves as an index into the EDT. It is the converse of the return unit name function.

Convert device type to look-up value - Function code 7

This function will convert a four-byte UCB device type to an internal representation of the unit name, to serve as an index into the EDT. The convert device type to look-up value function allows programs that have only a four-byte UCB device type to query the EDT. It may be used whenever a look-up value is required as input to the unit verification service.

Return attributes - Function code 8

This function returns general information about the specified unit name.

Check units with no validity bit - Function code 9

This function causes the check units function to set only a return code in case of an invalid device number. The no validity bit function saves processing in the case of a check units request for a single unit: because the parameter list is not modified, it need not be in key 1 storage. The function is available through IEFGB4UV or IEFAB4UV only.

Specify subpool for returned storage - Function code 10

This function is used with the return UCB addresses function or with the return unit names for a device class function. It allows you to specify a particular subpool to return the requested information in. This function is available through IEFEB4UV only.

Return unit names for a device class - Function code 11

This function returns a list of IBM generic device types (for example, 3390) and/or esoteric group names (for example, TAPE) associated with the input device class. This function is available through IEFEB4UV only.

Callers of IEFEB4UV

The unit verification routine, IEFEB4UV, is for both problem program callers and for authorized callers. It runs in task mode in the caller's key.

To use IEFEB4UV, the calling program must do the following:

- Create the input data structures and parameter list
- Place the address of an 18-word save area in register 13
- Provide a recovery environment
- Pass control to IEFEB4UV using the LINK and LINKX macro.

On return, IEFEB4UV restores all registers except register 15, which contains a return code.

Callers of IEFGB4UV or IEFAB4UV

The IEFGB4UV and IEFAB4UV routines are for authorized callers and run in task mode in key 1. To use the IEFGB4UV or IEFAB4UV unit verification routine, the calling program must do the following:

- Create the input data structures and parameter list in non-fetch protected key 1 storage
- Place the address of an 18-word save area in register 13
- Provide a recovery environment if authorized
- Pass control to IEFGB4UV or IEFAB4UV using the LINK macro

On return, the routine restores all registers except register 15, which contains a return code.

IEFAB4UV is the 24-bit addressing mode interface that is provided for compatibility with previous releases. It changes the environment to 31-bit addressing mode and calls IEFGB4UV to perform the requested functions. On return, it converts the environment back to 24-bit addressing mode before returning to the caller.

Input to and output from unit verification service routines

You must supply a two-word parameter list when invoking the unit verification routine (IEFGB4UV, IEFAB4UV, or IEFEB4UV).

The first word contains the address of a unit table. The contents vary according to the function(s) requested.

The second word contains the address of a 2 byte field (FLAGS), in which you specify the function(s) requested.

The bits in the FLAGS parameter field have the following meanings:

Bit	Function Requested
0	Check groups
1	Check units
2	Return unit name
3	Return UCB addresses
4	Return group ID
5	Indicate unit name is a look-up value
6	Return look-up value
7	Convert device name to a look-up value
8	Return attributes
9	Check units with no validity bit
10	Specify subpool for returned storage
11	Return unit names for a device class
12-15	Reserved for IBM use

Input parameter list

Figure 95 shows the input parameter list needed to invoke the unit verification service routine.

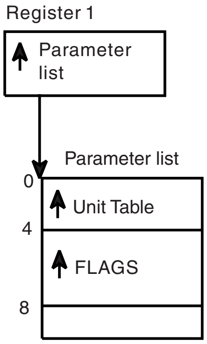


Figure 95. Input Parameter List

Input and output data structures

The diagrams on the following pages show the input data structures and parameters needed to invoke the unit verification service routine. The output data structure returned by the routine is also shown.

You must declare the structures exactly as shown to get the response indicated by the function(s) you request in FLAGS.

Because many of the input and output data structures are the same, you can request many of the functions in combinations with other functions. The following table lists the valid single functions and combinations of functions that you can request in a single invocation of the unit verification service.

Code	**Verification Service**
0	
0,1	
0,1,5	
0,1,5,9	IEFGB4UV or IEFAB4UV only
1	
1,5	
1,9	IEFGB4UV or IEFAB4UV only
1,5,9	IEFGB4UV or IEFAB4UV only
2	
2,7	
2,8	
2,7,8	
3	
3,5	
3,8	
3,10	IEFEB4UV only
3,5,7	
3,5,10	IEFEB4UV only
3,8,10	IEFEB4UV only
3,5,7,10	IEFEB4UV only
4	
6	
6,8	
7	
8	
10,11	IEFEB4UV only
11	IEFEB4UV only

Register 15 if request fails

On return, register 15 will contain a return code. If the invocation fails, it may be for one of the following reasons:

1. If you request a function that is not valid or a combination of functions that are not valid, register 15 contains a return code of 28 and the request fails.

2. If the JES control table (JESCT) does not contain valid pointers, the environment is incorrect. Register 15 contains a return code of 24. The request fails.

Requesting function code 0 (check groups)

Input: Set bit 0 in FLAGS to 1.

The input unit table structure is shown below.

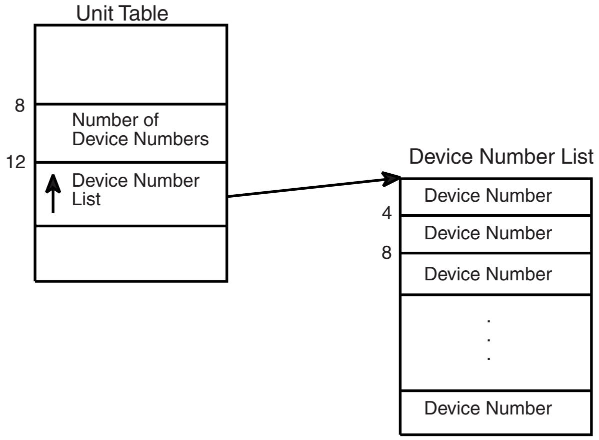


Figure 96. Requesting Function Code 0 (Check Groups)

Output: None.

Register 15 contains one of the following return codes:

Code	Meaning
0	The specified input is correct.
12	The device groupings are not valid.
28	The required input is not specified or is not valid.

Requesting function code 1 (check units)

Input: Set bit 1 in FLAGS to 1.

The input unit table structure is shown below.

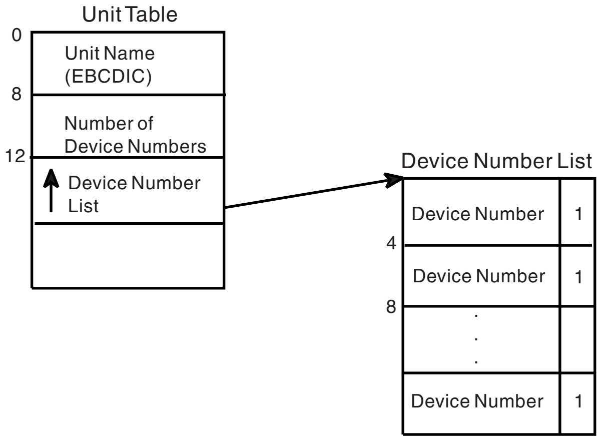


Figure 97. Requesting Function Code 1 (Check Units)

Output: If a device number is not valid, bit 0 of the FLAG byte is set to 1.

Register 15 contains one of the following return codes:

Code	Meaning
0	The specified input is correct.
4	The specified unit name is not valid.
8	Unit name has incorrect units assigned.
20	One or more device numbers are not valid.
28	The required input is not specified or is not valid.

Requesting function code 2 (return unit name)

Input: Set bit 2 in FLAGS to 1.

The input unit table structure is shown below.

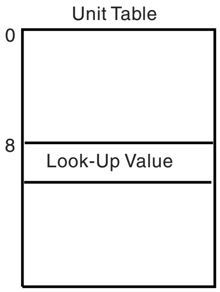


Figure 98. Requesting Function Code 2 (Return Unit Name)

Output: The unit table contains the unit name as shown in the following figure.

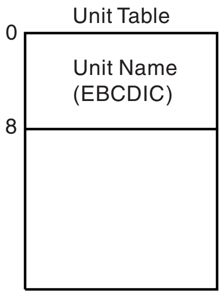


Figure 99. Output from Function Code 2 (Return Unit Name)

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit table contains the EBCDIC unit name.
4	The look-up value could not be found in the EDT.
28	The required input is not specified or is not valid.

Requesting function code 3 (return UCB addresses)

Input: Set bit 3 in FLAGS to 1.

The input unit table structure is shown below.

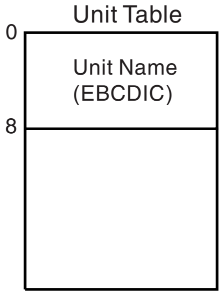


Figure 100. Requesting Function Code 3 (Return UCB Addresses)

Output: The unit table contains a pointer to the UCB Pointer List as shown in the following figure.

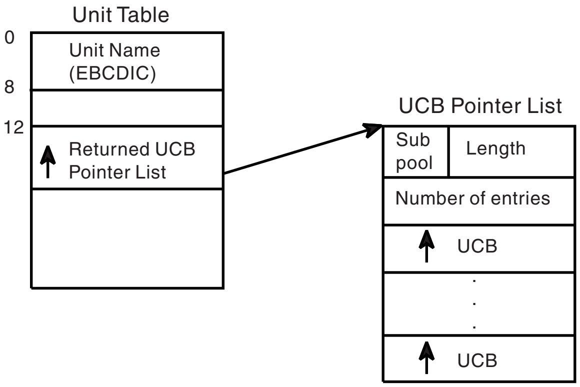


Figure 101. Output from Function Code 3 (Return UCB Addresses)

For authorized callers, the list is returned in the default subpool, 230. For unauthorized callers, the subpool default is 0. See function code 10 for a description of how to change the default subpool. The caller must free the number of bytes in the length field from the subpool before exiting.

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit table contains the pointer to the UCB pointer list.
4	The unit name could not be found in the EDT.
16	Storage was not available for the UCB pointer list.
28	The required input is not specified or is not valid.

Requesting function code 4 (return group ID)

Input: Set bit 4 in FLAGS to 1.

The input unit table structure is shown below.

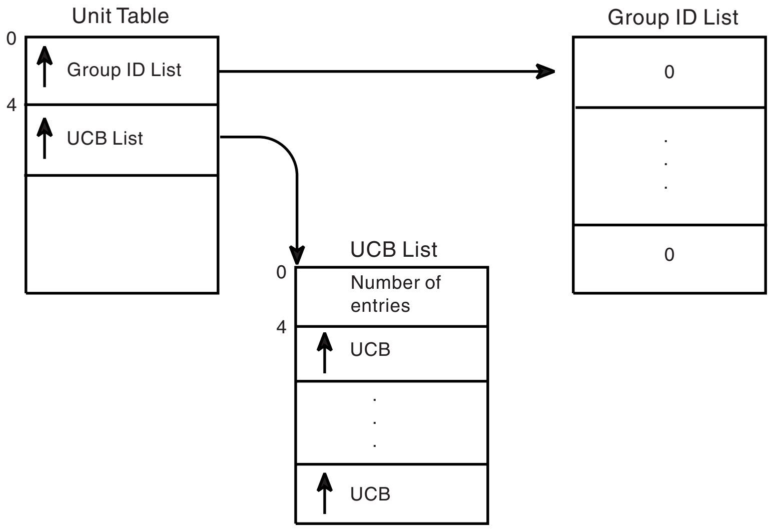


Figure 102. Requesting Function Code 4 (Return Group ID)

Note: One fullword is provided in the group id list for each UCB in the UCB list. Initialize all entries to zero.

Output: The group id list contains the group id corresponding to each UCB in the input UCB list.

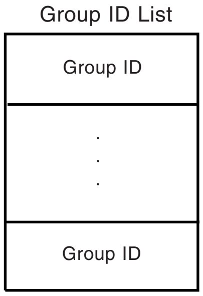


Figure 103. Output from Function Code 4 (Return Group ID)

Note: If the UCB is not in the EDT, the group id for that particular entry remains zero.

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
24	The JESCT does not contain valid pointers.

Requesting function code 5 (indicate unit name is a look-up value)

Input: Set bit 5 in FLAGS to 1.

The input unit table structure is shown below.

This function is not valid by itself. It must be used in combination with other functions that require a unit name as input. If you know the look-up value corresponding to the unit name, you can substitute it for the unit name in the input unit table. The following figure represents the first two fullwords of the unit table when function code 5 is requested.

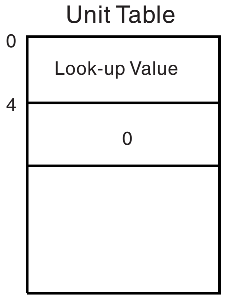


Figure 104. Requesting Function Code 5 (Indicate Unit Name is a Look-up Value)

Output: None specifically associated with this function.

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
4	The input look-up value could not be found in the EDT.
28	The required input is not specified or is not valid.

Requesting function code 6 (return look-up value)

Input: Set bit 6 in FLAGS to 1.

The input unit table structure is shown below.

This function is the opposite of the return unit name function (Code 2). The following figure represents the unit table structure when you request function code 6.

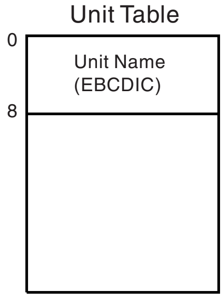


Figure 105. Requesting Function Code 6 (Return Look-up Value)

Output: The unit table contains the look-up value.

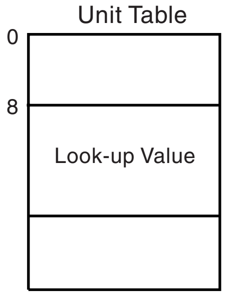


Figure 106. Output from Function Code 6 (Return Look-up Value)

Register 15 contains one of the following return codes:

Code	**Meaning**
0	Processing is successful.
4	The unit name could not be found; no look-up value is returned.
24	The JESCT does not contain valid pointers.

Requesting function code 7 (convert device type to look-up value)

Input: Set bit 7 in FLAGS to 1.

The input unit table structure is shown below.

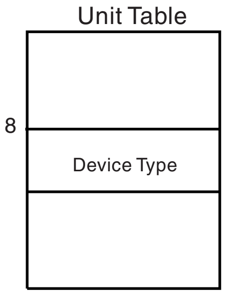


Figure 107. Requesting Function Code 7 (Convert Device Type to Look-up Value)

Note: The device type is in the format of the UCBTYP field of the UCB.

Output: The unit table contains the look-up value.

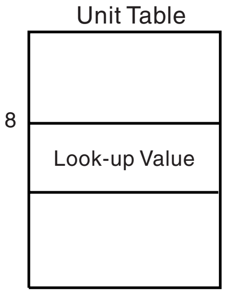


Figure 108. Output from Function Code 7 (Convert Device Type to Look-up Value)

The conversion of the device type to a look-up value is done in place. There is no error checking of the device type.

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
4	The input device type is not valid; no look-up value is returned.
28	The required input is not specified or is not valid.

Requesting function code 8 (return attributes)

Input: Set bit 8 in FLAGS to 1.

The input unit table structure is shown below.

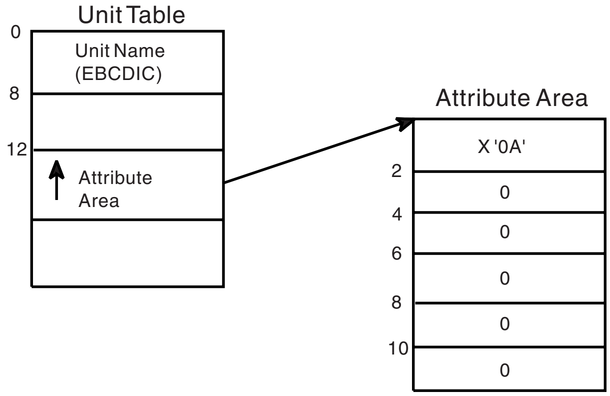


Figure 109. Requesting Function Code 8 (Return Attributes)

Output: The attribute area contains the following:

Byte **Contents**

0 Length of the attribute area (X'0A') This must be filled in prior to calling the unit verification service.

1-2 Flags describing the unit name:
- Bit 0 on — unit name is an esoteric group name
- Bit 1 on — unit name is VIO-eligible
- Bit 2 on — unit name contains 3330V units
- Bit 3 on — unit name contains TP class devices
- Bits 4-7 are not used.

3 Number of device classes in the unit name

4-7 Number of generic device types in the unit name

8-9 Reserved

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit name was found; the attributes are returned.
4	The unit name was not found; no attributes are returned.
28	The required input is not specified or is not valid.

Requesting function code 9 (check units with no validity bit)

The check unit with no validity bit function is available through IEFAB4UV and IEFGB4UV only.

Input: Set bit 9 in FLAGS to 1.

There is no other input associated with this function except the input for the check units function. This function must be used in combination with the check units function (code 1).

Output: See the output from the check units function.

The FLAG byte of the device number list is not altered; only the return code from the check units function (code 1) is available to determine if any device numbers in the device numbers list are not valid.

Requesting function code 10 (specify subpool for returned storage)

The specify subpool for returned storage function is available only through IEFEB4UV.

Input: Set bit 10 in FLAGS to 1. This function is not valid alone and must be used with either the return UCB addresses function (code 3) or the return unit name function for a device class (code 11). The input unit table structure is shown in the following figure.

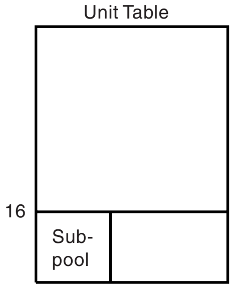


Figure 110. Requesting Function Code 10 (Specify Subpool for Returned Storage)

Output: See the output from the function that this is invoked in combination with.

The subpool field of the returned list contains the input subpool, and the returned list resides in that subpool. No error checking of the subpool is performed. If the subpool is not valid, the unit verification routine fails.

Requesting function code 11 (return unit names for a device class)

The return unit names for a device class function is available only through IEFEB4UV.

Input: Set bit 11 in FLAGS to 1.

The following figure shows the input unit table structure.

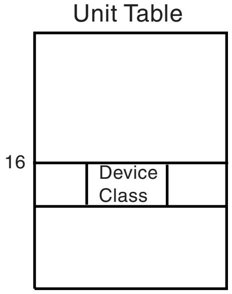


Figure 111. Requesting Function Code 11 (Return Unit Names for a Device Class)

Output: The unit table contains the pointer to the names list as shown in the following figure.

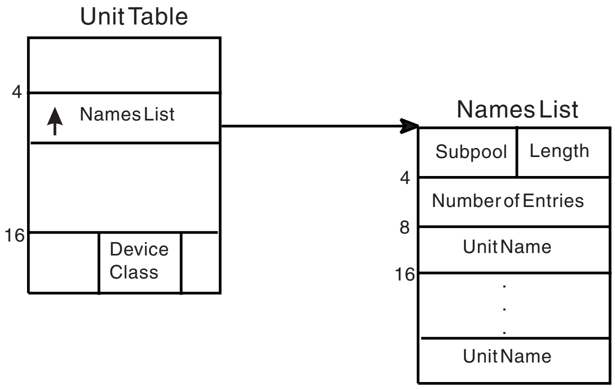


Figure 112. Output from Function Code 11 (Return Unit Names for a Device Class)

For authorized callers, the subpool that the names list is returned in is the default subpool 230. For unauthorized callers, the default subpool is 0. To change this default, see the description for function code 10 (specify subpool for returned storage). The caller must free the number of bytes in the length field from the subpool before exiting.

Register 15 contains one of the following return codes:

Code	Meaning
0	The pointer to the names list is stored in the unit table.
16	Storage was not available for the names list.
28	The required input is not specified or is not valid.

Requesting multiple functions - Examples

The following examples show the input to and output from multiple functions.

- Example 1 shows the multiple functions of codes 0 and 1.
- Example 2 shows the multiple functions of codes 3 and 10.
- Example 3 shows the multiple functions of codes 1 and 5.

Example 1 - Function codes 0 and 1

Input:

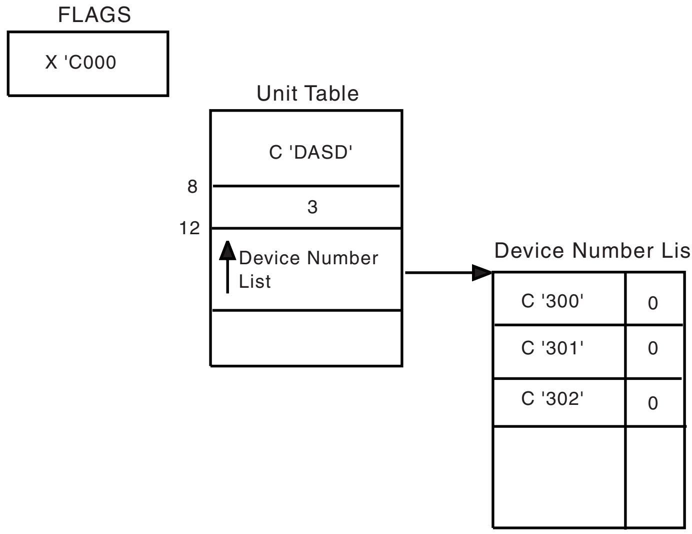


Figure 113. Input for Function Codes 0 and 1

Output:

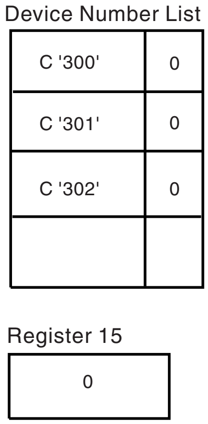


Figure 114. Output from Function Codes 0 and 1

Note: All input device numbers make up a single allocation group and are associated with the esoteric unit name DASD.

Example 2 - Function codes 3 and 10

Input:

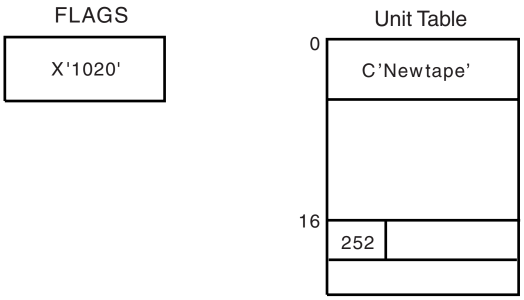


Figure 115. Input for Function Codes 3 and 10

Output:

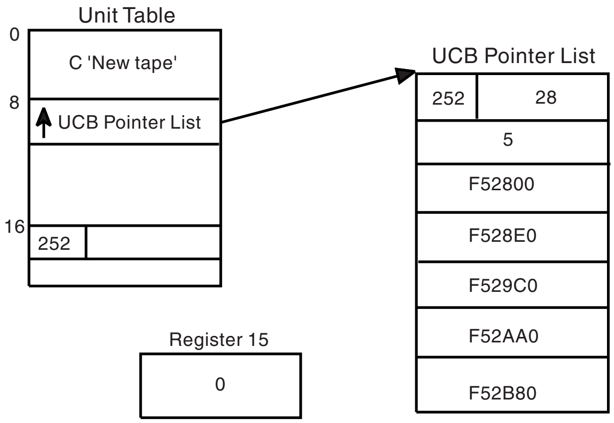


Figure 116. Output from Function Codes 3 and 10

Note: The caller must be authorized to request subpool 252. The unit verification service invoked must be IEFEB4UV because function code 10 is requested. The caller must free the UCB pointer list before exiting.

Example 3 - Function codes 1 and 5

Input:

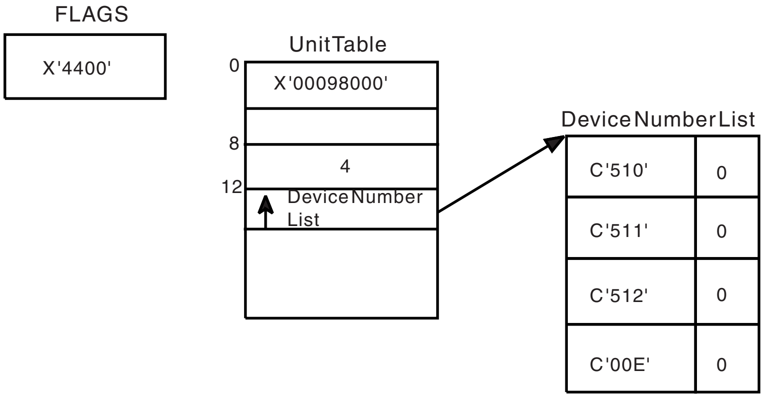
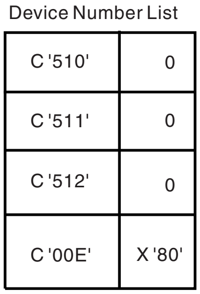


Figure 117. Input for Function Codes 1 and 5

Output:



Device Number List

C '510'	0
C '511'	0
C '512'	0
C '00E'	X '80'

Figure 118. Output from Function Codes 1 and 5

Note: Device 00E did not belong to the unit name that was associated with the input look-up value.

Appendix B. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, `3` or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the

default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

`* indicates an optional syntax element that is repeatable`
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

`+ indicates a syntax element that must be included`
The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (http://www.ibm.com/software/support/systemsz/lifecycle/)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This information is intended to help the customer to code macros that are available to authorized assembler language programs.

This information primarily documents intended programming interfaces that allow the customer to write programs to obtain services of z/OS.

This book also documents information that is NOT intended to be used as programming interfaces of z/OS. This information is identified where it occurs, either by an introductory statement to a chapter or section, or by the following marking:

NOT Programming Interface information

End of NOT Programming Interface information

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 (http://www.ibm.com/legal/copytrade.shtml).

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Product Number: 5650-ZOS

Printed in USA

SA23-1371-02