GC28-1150-4
File No. S370-36

Program Product

# MVS/Extended Architecture System Programming Library: System Macros and Facilities Volume 1

**MVS/System Product:**

| | |
|---|---|
| **JES3 Version 2** | **5665-291** |
| **JES2 Version 2** | **5740-XC6** |

IBM

**Fifth Edition (September, 1989)**

This is a major revision of, and obsoletes GC28-1150-3 and Technical Newsletter GN28-1901. See the Summary of Amendments following the Contents for a summary of the changes made to this manual. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This edition applies to Version 2 Release 2 of MVS/System Product program number 5665-291 or 5740-XC6 and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are made periodically to the information herein; before using this publication, in connection with the operation of IBM systems, consult the latest *IBM System/370 Bibliography,* GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product in this publication is not intended to state or imply that only IBM's product may be used. Any functionally equivalent product may be used instead. This statement does not expressly or implicitly waive any intellectual property right IBM may hold in any product mentioned herein.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department D58, Building 921-2, PO Box 950, Poughkeepsie, New York 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

# Preface

This two-volume publication describes supervisor and scheduler facilities that the system programmer can use. In this publication, a system programmer is defined as a programmer whose programs run in supervisor state, system key 0-7 or access APF-authorized libraries. The publication included the macro instructions and parameters used to obtain the functions.

Volume 1 contains descriptions of the supervisor and scheduler services available to a system programmer. Most of the services described are supervisor services; however, the scheduler functions available through the use of the DYNALLOC macro instruction are also described. Volume 1 includes a description of the DYNALLOC macro instruction. Some of the topics discussed in Volume 1 are also discussed in *Supervisor Services and Macro Instructions*; however in Volume 1, these topics are extended to include functions that are restricted to system programmers or used primarily by system programmers.

Volume 2, GC28-1151, contains the formats and descriptions of the supervisor macro instructions. Volume 2 provides system programmers with the information necessary to code the macro instructions. Each macro instruction is completely described, in Volume 2, but restrictions, requirements, and environmental considerations for the effective use of each macro is explained in Volume 1.

## Trademarks

The following are trademarks of International Business Machines Corporation.

- MVS/ESA™
- MVS/DFP™
- MVS/SP™
- MVS/XA™

## Related Publications

*A Structured Approach to Describing and Searching Problems*, SC34-2129

*The Considerations of Physical Security in a Computer Environment*, G520-2700

*Data Security Controls and Procedures - A Philosophy for DP Installations*, G320-5649

*MVS/ESA Checkpoint/Restart User's Guide*, SC26-4503

*MVS/ESA Data Administration: Macro Instruction Reference*, SC26-4506

*MVS/ESA Data Facility Product Version 3: Diagnosis Reference*, LY27-9551

*MVS/ESA Linkage Editor and Loader User's Guide*, SC26-4510

*MVS/ESA System-Data Administration*, SC26-4515

*MVS/ESA VSAM Administration Guide*, SC26-4518

*MVS/Extended Architecture Checkpoint/Restart User's Guide*, GC26-4139

*MVS/Extended Architecture Data Administration: Macro Instruction Reference*, GC26-4141

*MVS/Extended Architecture Data Facility Product Version 2: Diagnosis Reference*, LY27-9530

*MVS/Extended Architecture Debugging Handbook Volume 1*, LC28-1164

*MVS/Extended Architecture Debugging Handbook Volume 2*, LC28-1165

*MVS/Extended Architecture Debugging Handbook Volume 3*, LC28-1166

*MVS/Extended Architecture Debugging Handbook Volume 4*, LC28-1167

*MVS/Extended Architecture Debugging Handbook Volume 5*, LC28-1168

*MVS/Extended Architecture Debugging Handbook Volume 6*, LC28-1169

*MVS/Extended Architecture Diagnostic Techniques*, LY28-1199

*MVS/Extended Architecture Interactive Problem Control System Planning and Customization*, GC28-1406

*MVS/Extended Architecture Interactive Problem Control System User's Guide and Reference*, GC28-1297

*MVS/Extended Architecture Job Control Language User's Guide*, GC28-1351

*MVS/Extended Architecture Job Control Language Reference*, GC28-1352

*MVS/Extended Architecture Linkage Editor and Loader User's Guide*, GC26-4143

*MVS/Extended Architecture Message Library: System Messages Volume 1*, GC28-1376

*MVS/Extended Architecture Message Library: System Messages Volume 2*, GC28-1377

*MVS/Extended Architecture Operations: System Commands*, GC28-1206

*MVS/Extended Architecture Planning: Global Resource Serialization*, GC28-1062

*MVS/Extended Architecture Supervisor Services and Macro Instructions*, GC28-1154

*MVS/Extended Architecture System-Data Administration*, GC26-4149

*MVS/Extended Architecture System Initialization Logic*, LY28-1200

*MVS/Extended Architecture System Logic Library: Allocation/Unallocation*, LY28-1615

*MVS/Extended Architecture System Programming Library: Initialization and Tuning*, GC28-1149

*MVS/Extended Architecture System Programming Library: Service Aids*, GC28-1159

*MVS/Extended Architecture System Programming Library: System Macros and Facilities Volume 2*, GC28-1151

*MVS/Extended Architecture System Programming Library: System Modifications*, GC28-1152

*MVS/Extended Architecture System Programming Library: User Exits*, GC28-1147

*MVS/Extended Architecture System Programming Library: 31-Bit Addressing*, GC28-1158

*MVS/Extended Architecture VSAM Administration Guide*, GC26-4151

*OS/VS Mass Storage System Extensions Messages*, SH35-0041

*Resource Access Control Facility (RACF): General Information Manual*, GC28-0722

*Security Assessment Questionnaire*, GX20-2381

*System Programming Library: RACF*, SC28-1343

*TSO Extensions Version 2 Programming Guide*, SC28-1874

*TSO Extensions Version 2 Programming Services*, SC28-1875

*TSO Extensions Version 2 System Programming Command Reference*, SC28-1878

*370-Extended Architecture: Principles of Operation*, SA22-7085

**Notes:**

1. All references to RACF in this publication indicate the program product Resource Access Control Facility (5740-XXH).
2. All references to Assembler H in this publication indicate the program product Assembler H Version 2 (5668-962).
3. All references to RMF in this publication indicate the program product Resources Measurement Facility (5665-274)

# Contents

# Figures

# Summary of Amendments

**Summary of Amendments**
**for GC28-1150-4**
**MVS/System Product Version 2 Release 2.3**

This major revision contains changes to support MVS/System Product Version 2 Release 2.3. Changes include:

- MVS/XA support for MVS/Data Facility Product Version 3 Release 1.0, which introduces the storage management subsystem (SMS). SMS provides new function for data and storage management.

  In this book, "with SMS" indicates information that applies when SMS is installed and active; "without SMS" indicates SMS is not installed or is not active.

- Changes to the dynamic allocation service.

Maintenance changes and other documentation enhancements include:

- Addition of documentation related to the TIMEUSED macro.
- Re-write of section on "Inter-Address Space Communication".
- Re-write of section on "Managing SWA Control Blocks", now called "Accessing the Scheduler Work Area".
- Changed information related to using the SDUMP macro instruction and obtaining a summary dump.
- Changed information related to using the LOAD macro instruction.
- Addition of a section on "Protecting the Vector Facility".
- Changed information related to the extended ECB.
- Other minor technical and editorial changes throughout.

**Summary of Amendments**
**for GC28-1150-3**
**MVS/System Product Version 2 Release 2**

This major revision describes how to use the SYMREC macro, which is new. It also describes changes to the DATOFF, DOM, VSMLOC, WTO, and WTOR macros, and changes that affect:

- The use of address space by PC/AUTH.
- User-written SVC routines.
- The allocation default module (IEFAB445).
- Dumps obtained via SDUMP.
- The jobstep DD limit.

**Summary of Amendments**
**for GC28-1150-2**
**MVS/System Product Version 2 Release 1.3**

This major revision contains information about the new macro, IOSINFO, in support of System Product Version 2 Release 1.3 and minor technical and editorial changes.

# Introduction

The system facilities described in this publication include both supervisor and scheduler services. The supervisor services provide the resources that your programs need while assuring that as many of these resources as possible are being used at a given time. The scheduler services described in this publication are the scheduler functions that are available through the use of the dynamic allocation macro instruction (DYNALLOC). Knowing the conventions and characteristics of the system facilities will help you to design more efficient programs.

*Volume 1* describes those supervisor services that should be restricted in use to systems programmers and installation-approved personnel. If a particular topic includes a description of a macro instruction, the macro instruction is given in parentheses after the topic heading. *Volume 1* includes a description of the DYNALLOC macro instruction. The supervisor macros and parameters are described in *Volume 2*. The topics described in *Volume 1* are:

**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 supervisor 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.

**Reporting System Characteristics:** Collecting information about resources and their requestors and using the SRM and SYMREC reporting interfaces are described in this topic.

**Communication:** This topic is divided into four distinct and different types of communication. These are:

- Interprocessor communication available through the use of the SIGP instruction
- Communication with the operator available through the use of the WTO, WTOR, and DOM macro instructions
- Asynchronous inter-address space communication available through the use of the SCHEDULE macro instruction
- Synchronous inter-address space communication available through the use of cross memory facilities

**Virtual Storage Management:** Virtual storage allows you to write large programs without the need for complex overlay structures. This topic describes how to allocate and free virtual storage. It also includes descriptions of the VSM functions, available through the use of the VSMLIST, VSMLOC, and VSMREGN macro instructions, and a description of managing SWA control blocks.

**Real Storage Management:** The supervisor administers the use of real storage and directs the movement of virtual pages between auxiliary storage and real storage in page size blocks. The services provided allow you to release virtual storage contents, load virtual storage areas into real storage, and page out virtual storage areas from real storage.

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

**Normal and Abnormal Program Termination:** The supervisor provides facilities for writing exit routines to handle specific types of interruptions. It is not likely, however, that you will be able to write routines to handle all types of abnormal conditions. The supervisor therefore provides for termination of your program when you request it by issuing an ABEND macro instruction or when the control program detects a condition that will degrade the system or destroy data. This topic describes the recovery termination manager, system trace facilities, recovery routines, the use of the SPIE/ESPIE macro instructions to process program interruptions, the use of the SLIP command to intercept errors, and the use of the SDUMP macro to obtain a dump of virtual storage.

**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:** Two types of exit routines are described in this topic. They are asynchronous exit routines and timer disabled interrupt exits.

**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.

**UCB Scan Services:** This topic describes the function of the UCB scan routine (IOSVSUCB). This routine allows you to scan each unit control block (UCB) in the system or in a specified device class.

**Dynamic Allocation (SVC 99) Services:** This topic describes the functions provided by dynamic allocation (SVC 99). A description of the parameter list used to request SVC 99 functions, the SVC 99 return codes, error codes, and information codes are included.

# 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.

Most of the information concerning subtask creation and control appears in *Supervisor Services and Macro Instructions*. This chapter continues discussion in the following areas:

- Task creation (ATTACH macro instruction)
- Issuing an internal START command (MGCR macro instruction)
- Communicating with a problem program (EXTRACT and QEDIT macro instructions)
- Changing the priority of a task (CHAP macro instruction)

## Creating a New Task (ATTACH)

The ATTACH macro instruction causes the control program to create a new task. The complete use of the macro instruction is described in *Supervisor Services and Macro Instructions.*

The macro instruction has parameters that provide the authorized user (protection key 0-7 or supervisor state) flexibility in using the macro instruction's services. If authorized tasks do not specify a particular parameter, the default value for that parameter is assigned. These defaults include:

- JSTCB=NO -- the attached task is a task in the present job step.
- SM=PROB -- the new task is to run in problem program mode.
- SVAREA=YES -- a save area is needed for the new task.
- KEY=PROP -- the protection key of the newly created task is to be the same as the task using ATTACH.
- DISP=YES -- the subtask is to be dispatchable.
- TID=0 -- the task identifier of the new task is 0.
- JSCB -- omission of this parameter specifies that the job step control block of the attaching task is also used for the new task.
- NSHSPV and NSHSPL -- omission of these parameters specifies that subpools 236 and 237, if they exist, are to be shared with the subtask.
- RSAPF=NO -- The APF authorization of the step is to be unchanged.

### Changing the Defaults of ATTACH

Rather than accepting the default values, (assuming the task is authorized), you can extend the facilities of the ATTACH macro instruction by coding the following values:

- JSTCB=YES -- the attached task is a new job step task. In this case, the address of the TCB of the newly created task is placed in the TCBJSTCB field of the attached TCB. The initiator attaches the first load module of a job as a job step task. For such an attach, the program manager does not search the job library of the attaching task.

  Also, only under a job step task can a system program (system key or supervisor state) attach a load module from a non-system library.

  In order to attach a job step task, the attaching task (and any of its subtasks) must be job step tasks. If one of these conditions is not met, the new task will not be created.

- SM=SUPV -- the system is to run in supervisor mode when executing the attached task.

  Supervisor state is a requirement for issuing privileged instructions (for example, LPSW). You can specify supervisor mode via this parameter or via the MODESET macro instruction.

- SVAREA=NO -- the new task does not need a save area.

  The save area is obtained from the user's region. Because it might not always be desirable to have a save area (for example, the user's region might not be defined at the time of a system ATTACH), this parameter can be used to specify that no save area is to be created.

- KEY=ZERO -- the protection key of the newly created task is zero.

  Protection key zero allows the new task to reference any defined storage and pass all validity checks.

- DISP=NO -- the subtask is to be nondispatchable.

  This parameter causes the primary nondispatchability bit TCBANDSP to be turned on in the new TCB. As a result, the new TCB will not be dispatched. Thus, specifying DISP=NO allows the originating task to alter the new TCB. The new task remains nondispatchable until the originating task issues the STATUS macro instruction with the RESET option to reset TCBANDSP.

  **Note:** STATUS START TCB will not make the new TCB dispatchable.

- TID=task id -- the task identifier specified is to be placed in the TCBTID field of the attached task.

  The task identifier can be set to identify critical system tasks. Other uses of this parameter are not recommended.

- JSCB=job step control block address -- the address specified for the JSCB is to be used for the new task.

  This parameter sets the TCBJSCB to the address of a job step control block. This action, normally associated with the creation of a job step task, is not required by ATTACH.

- NSHSPV=subpool number and NSHSPL=subpool list address -- subpools 236 and 237 are not to be shared with the new task.

  Subpools 236 and 237 are known as the scheduler work area (SWA). This parameter allows the scheduler to control these subpools.

- RSAPF=YES -- reset the step APF authorization.

  This parameter allows a system program that is not running APF authorized to ATTACH a subtask and have the APF authorization for the step reset according to the attributes of the subtask. The subtask must be attached while in the problem program state and must be in a non-system key. For more information on this parameter see "Authorization Results Under Various Conditions" in the "Protecting the System" section.

## Issuing an Internal START or REPLY Command (MGCR)

A program can issue an internal START or REPLY command using the MGCR macro instruction and can pass 31 bits of information, called a token, to the program being started (in the case of the START command). An internal REPLY command is available to reply to a WTOR message. Before issuing the MGCR macro instruction, initialize a buffer for the command and the token, if any, as follows:

| 1 byte | 1 byte | 2 bytes | variable length | 4 bytes |
|---|---|---|---|---|
| flags1 | length | flags2 | text | 31 bit token right justified |

You must also set register 0 to zero before issuing the MGCR macro instruction.

**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 token field.

**flags2**
If a token is present, flags2 must be set to X'0800', otherwise, it must be set to X'0000'.

**text**
The text field contains the START or REPLY command followed by operands and, optionally, comments.

**token**
This field contains any desired information to be communicated to the started program. Token is meaningful only for the START command.

Figure 1 shows how the buffer is set up.

The IEZMGCR mapping macro, in SYS1.MACLIB, is available to map the buffer.

```
          SR   REG0,REG0        INDICATE SYSTEM ISSUED
*                               COMMAND
MGCRMAC   MGCR MGRCDATA
          .
          .
MGCRDATA  EQU  *
FLG1      DC   X'80'
LGTH      DC   AL1(TOKEN-MGCRDATA)
FLG2      DC   X'0800'
TXT       DC   C'S IMS ***ANY COMMENTS***'
TOKEN     DC   AL4(ECB)         ECB ADDR
```

Figure 1. Setting Up the Buffer for MGCR

# Communicating with a Problem 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 instruction 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 instruction is written as follows, and returns what is indicated in Figure 2.

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

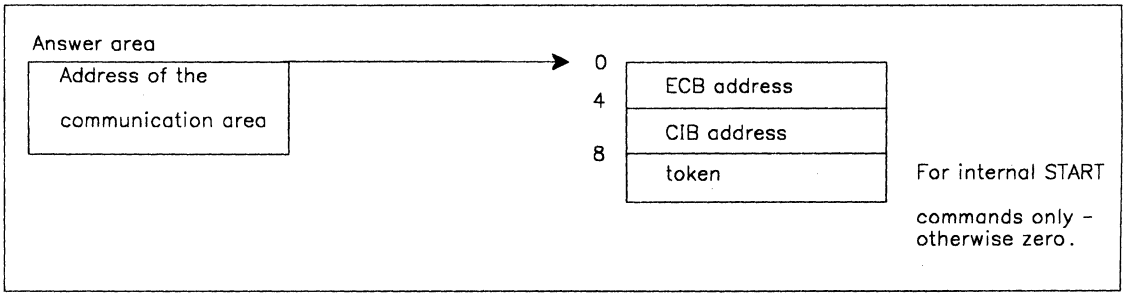



Figure 2. EXTRACT ECB, CIB Pointers, and Token

The CIB contains the information specified on the STOP, START, or MODIFY command, as shown in Figure 3. If the job was started from the console, the EXTRACT macro instruction will point to the START CIB. If the job was not started from the console, the address of the first CIB will be zero.

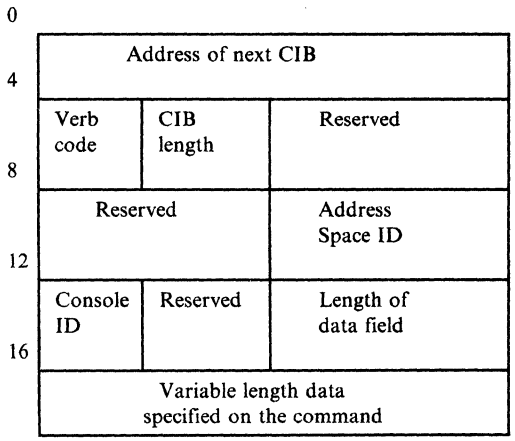



Figure 3. Command Input Buffer Contents

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

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

**Notes:**

1. The address of the pointer to the CIB is the contents of the answer area plus 4 bytes, as shown in Figure 2.
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 instruction:

```
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.

**Note:** When using the address or addresses returned from the EXTRACT macro as input to the QEDIT macro, you must establish addressability via the IEZCOM mapping macro, in SYS1.MACLIB, based on the address returned by the EXTRACT.

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 instruction 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 instruction, the communications ECB, and all CIBs are in protected storage and may not be altered.

The program in Figure 4 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.

```
QEDITEX  CSECT
         BALR  12,0                PROGRAM...
         USING *,12                ...ADDRESSABILITY
*
* INITIALIZATION PROCESSING - DELETE START CIB
*
* OBTAIN ADDRESS OF CIB
*
         LA    5,ANSRAREA          ADDRESS OF RESPONSE AREA FOR QEDIT
         EXTRACT (5),FIELDS=COMM   OBTAIN ADDRESS OF THE                X
                                   COMMUNICATIONS AREA FOR THE          X
                                   CURRENT TASK
*
         L     5,ANSRAREA          LOAD ADDRESS OF COMMUNICATIONS AREA
         USING COMLIST,5           ESTABLISH ADDRESSABILITY TO IEZCOM
         L     3,COMCIBPT          OBTAIN ADDRESS OF CIB
         LTR   3,3                 WAS A CIB ADDRESS RETURNED?
         BZ    SETCOUNT            NO, CONTINUE INITIALIZATION
         USING CIBNEXT,3           ESTABLISH ADDRESSABILITY TO IEZCIB
*
* MOVE DATA FROM CIB TO WORKING STORAGE
*
         LH    4,CIBDATLN          OBTAIN LENGTH OF DATA FIELD
         BCTR  4,0                 DECREASE LENGTH BY ONE
         EX    4,DATAMOVE          MOVE DATA TO WORKING STORAGE
*
* FREE THE START CIB, IF PRESENT
*
         CLI   CIBVERB,CIBSTART    FIRST CIB FOR START COMMAND?
         BNE   SETCOUNT            NO, CONTINUE INITIALIZATION
         QEDIT ORIGIN=COMCIBPT,BLOCK=(3) YES, FREE IT
         LTR   15,15               CHECK RETURN CODE
         BZ    SETCOUNT            IF RETURN CODE IS ZERO, THE CIB      X
                                   WAS FREED, CONTINUE
         WTO 'START CIB NOT FREED' IF RETURN CODE IS NOT ZERO,          X
                                   NOTIFY THE OPERATOR THAT             X
                                   THE CIB WAS NOT FREED
```

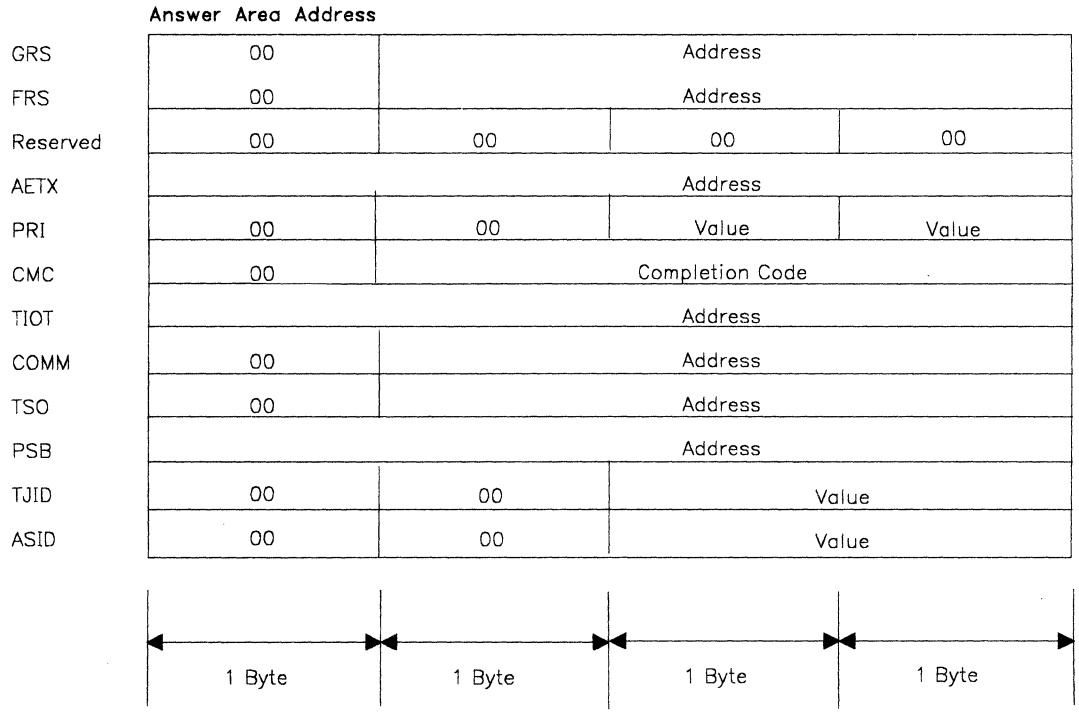
Figure 4 (Part 1 of 2). Example Using the EXTRACT and QEDIT Macros

```
*
* SET THE LIMIT ON MODIFY COMMANDS
*
SETCOUNT  EQU     *
          QEDIT ORIGIN=COMCIBPT,CIBCTR=2 SET LIMIT TO 2
*
* COMMAND PROCESSING LOOP
*
* CHECK THE COMMUNICATIONS ECB
*
WAIT      EQU     *
          L       4,COMECBPT          OBTAIN ADDRESS OF COMMUNICATIONS ECB
          WAIT    ECB=(4)             WAIT FOR STOP OR MODIFY COMMAND
*                                     NOTE: QEDIT CLEARS THE POSTED ECB
*                                     WHEN THE LAST CIB IS FREED
          L       3,COMCIBPT          OBTAIN ADDRESS OF CIB
*
* TEST FOR MODIFY COMMAND CIB
*
          CLI     CIBVERB,CIBMODFY    IS IT FOR A MODIFY COMMAND?
          BNE     TSTSTOP             NO, GO TEST FOR STOP
*
* PROCESS THE MODIFY COMMAND
*
* FREE THE MODIFY CIB
*
          QEDIT ORIGIN=COMCIBPT,BLOCK=(3) FREE IT
          B       WAIT                WAIT FOR NEXT COMMAND
*
* TEST FOR STOP COMMAND CIB
*
TSTSTOP   EQU     *
          CLI     CIBVERB,CIBSTOP     IS IT FOR A STOP COMMAND?
          BNE     ERROR1              NO, GO PROCESS AS ERROR
*
* PROCESS THE STOP COMMAND
*
* FREE THE STOP CIB
*
          QEDIT ORIGIN=COMCIBPT,BLOCK=(3) FREE THE CIB
          B       WAIT                WAIT FOR NEXT COMMAND
*
* ERROR HANDLER FOR UNRECOGNIZED CIB TYPE
*
ERROR1    EQU     *
*
* CONSTANTS AND DATA AREAS
*
          DS      0F
DATAAREA  DS      4F                  WORK AREA FOR CIB DATA
ANSRAREA  DS      F                   ANSWER AREA FOR EXTRACT MACRO
DATAMOVE  MVC     DATAAREA(0),CIBDATA MOVE DATA FROM CIB TO DATAAREA
          DSECT
          IEZCOM                      MAPPING MACRO FOR COMMUNICATION AREA
          DSECT
          IEZCIB                      MAPPING MACRO FOR CIB
          END
```

Figure 4 (Part 2 of 2). Example Using the EXTRACT and QEDIT Macros

## Providing an EXTRACT Answer Area

The EXTRACT macro instruction provides TCB information for either the active task or one of its subtasks. Figure 5 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.




**Note:** See the syntax of the EXTRACT macro instruction in Volume 2 for a description of these fields.

Figure 5. 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 5. (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.

## Changing the Priority of a Task (CHAP)

Programs should not use priority or precedence as a serialization mechanism because they become sensitive to changes in the system's dispatching algorithms. For example, the CHAP macro instruction does not ensure that tasks are dispatched in the expected order, due to dispatching on more than one processor. Also, the PRIORITY and DPRTY JCL parameters cannot be used to accomplish serialization. First, the system resources manager might change the dispatching priority of a task or job, allowing it to execute before a task with a previously higher priority. Second, because tasks can execute on more than one processor, tasks of different priority might be executed on more than one processor simultaneously.

# Program Management

You can specify whether you want a program loaded into storage above or below the 16 megabytes line and if you want a program loaded at a specific address. This information along with a description of synchronous exits, the use of checkpoint restart, and the use of re-entrant modules, is described in this chapter.

Load module structures, methods of passing control between programs, and the use of associated macro instructions are described in *Supervisor Services and Macro Instructions.*

## Residency and Addressing Mode of Programs

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

When a program is executing in 24-bit addressing mode, the system treats both instruction and data addresses as 24-bit addresses. This allows programs executing in 24-bit addressing mode to address 16 megabytes (16,777,216 bytes) of storage. Similarly, when a program is executing in 31-bit addressing mode, the system treats both instructions and data addresses as 31-bit addresses. This allows a program executing in 31-bit addressing mode to address 2 gigabytes (2,147,483,648 bytes or 128x16 megabytes) of storage. *SPL: 31-Bit Addressing* provides detailed information concerning the AMODE and RMODE of modules.

You can define the residency mode and the addressing mode of a program in the source code. Figure 6 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.

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

Figure 6. Assembler Definition of AMODE/RMODE

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 *Linkage Editor* 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. MVS/XA supports programs that execute in either 24-bit or 31-bit addressing mode.

The AMODE attribute can have one of the following values:

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

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

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

Information about the addressing mode as it applies to macro instructions can be found in Volume 2 under the topic "Addressing Mode and the Macro Instructions"

# Specifying Where the Module is to be Loaded (LOAD)

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 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=YES (the default) requests deletion at task termination. EOM=NO requests deletion at address space termination.

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 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. 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)

In general, the SYNCH macro instruction is used when a control 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 control program. This facility should be used only by system programmers or other installation-approved personnel. The program to which control is given must be in virtual storage when the macro instruction is issued. To ensure that a program receives control with a program key mask (PKM) consistent with its key, SYNCH processing forms the PKM using the default key in the TCB along with the key specified by the KEYADDR parameter. If the KEYMASK parameter is coded, the PKM formed thus far is ORed with the specified keymask.

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 macro instruction.

The SYNCH macro instruction is similar to the BALR instruction in that you can use register 15 for the entry point address.

SYNCH 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 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.

Label processing as a result of an OPEN macro instruction is an example of the use of the SYNCH macro instruction. Label processing might proceed to a point at which a user's processing program indicates that it wants or needs private processing. The control program's open routine would then issue a SYNCH macro instruction giving the address of the subroutine required for the user's private label processing.

## 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 (for example, a routine running in SRB, disabled, or cross memory mode) 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.
- Routines that use both PC/AUTH facilities and checkpoint/restart must reestablish their PC/AUTH environment at restart time. In addition, they must not use any PC/AUTH data (for example, a PC number) that was obtained before the restart.

- Subsystems that use the TCB subsystem affinity service cannot issue checkpoints. This is because the subsystem affinity table (SSAT) index values might change from one system initialization to another.

For additional information concerning the restrictions and use of the checkpoint/restart facility see *Checkpoint/Restart User's Guide*.

## Using Re-entrant Modules

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

- Two tasks in the same address space 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 macro instructions is to use the LIST and EXECUTE forms of the macro instructions with a dynamically acquired parameter list.

# Serialization

Planning is required when more than one program uses a serially reusable resource. A serially reusable resource is a resource that can be used by another program after the current use has been concluded; that is, a resource that should not be used or modified by more than one program within a given span of processing. Planning is also required when portions of some tasks depend on the completion of events in other tasks.

This chapter discusses some of the services available to control resources, and thus to help you plan ahead for a more efficient installation. The services discussed include:

- Locking (SETLOCK macro instruction)
- Must-complete function (ENQ and DEQ macro instructions)
- Shared DASD (RESERVE and EXTRACT macro instructions)
- Event completion (POST, SPOST, and EVENTS macro instructions)

Global resource serialization (ENQ, DEQ, RESERVE, or GQSCAN macro instructions) is another form of serialization available to an installation. This topic appears in *Supervisor Services and Macro Instructions* and *Planning: Global Resource Serialization.*

## When Resource Serialization Is Needed

Resource serialization is used to prevent a program from altering the content or status of a resource while another program is using that resource or is dependent on the content or status of that resource remaining unchanged for a given span of processing. For example, resource serialization prevents a program from issuing an SVC and changing the content of a control block while another SVC is using that control block.

### Serialization Requirements

It is necessary to determine and keep track of resources that must be serialized and the routines that access such resources. The only safe method of serialization is one of the following: ENQ/DEQ, WAIT/POST/EVENTS, SUSPEND/RESUME, locking at the TCB level, CS (compare and swap instruction), CDS (compare double and swap instruction), and TS (test and set instruction). Such forms of serialization are required in the following cases:

- Scanning of the command input buffer (CIB) chain. You could use the QEDIT macro instruction to manipulate the CIB chain.
- Using data in subpools shared between tasks.
- Using data referenced by more than one task. (For example, attached tasks can execute at the same time as the attaching task on different processors.)
- Referencing system control block fields that dynamically change after IPL. The serialization technique in this case must match that used by the system. (See the *Debugging Handbook* for information concerning the serialization requirements for a particular system control block.) Also, bits within a byte all require the same serialization technique.
- Accessing of data sets shared between tasks in the same address space, if the tasks update the data and if the access method is not VSAM or BDAM.
- Referencing any common data between an ESTAE exit and asynchronous exits, if ESTAE with ASYNCH = YES is issued.

# 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 lock manager controls a hierarchical locking structure with multiple types of locks to synchronize the use of serially reusable resources. The lock manager 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 the lock manager is restricted to key 0 programs running in supervisor state. 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. (For example, a unit control block is protected by a global lock because it relates to the entire system. Also, a system-related GETMAIN for a global subpool requires a global lock.)
- 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. (For example, an address space-related GETMAIN for a user subpool requires a local lock.)

All of the locks described in Figure 7, with the exception of the LOCAL and CML 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. Figure 7 summarizes the characteristics of MVS/XA locks.

| lock | global | local | spin | suspend | single | multiple (class) | shared/ exclusive |
|---|---|---|---|---|---|---|---|
| RSMGL | X | | X | | | X | |
| VSMFIX | X | | X | | X | | |
| ASM | X | | X | | | X | |
| ASMGL | X | | X | | | X | |
| RSMST | X | | X | | | X | |
| RSMCM | X | | X | | | X | |
| RSMXM | X | | X | | | X | |
| RSMAD | X | | X | | | X | |
| RSM | X | | X | | X | | X |
| VSMPAG | X | | X | | X | | |
| DISP | X | | X | | X | | |
| SALLOC | X | | X | | X | | |
| IOSYNCH | X | | X | | | X | |
| IOSUCB | X | | X | | | X | |
| SRM | X | | X | | X | | |
| TRACE | X | | X | | X | | X |
| CPU | X | | X | | | X | |
| CMS | X | | | X | X | | |
| CMSEQDQ | X | | | X | X | | |
| CMSSMF | 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; a user can obtain the CPU lock while holding any spin 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.

Figure 7. Summary of Locking Characteristics

## 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. Shared/exclusive locks are a category of spin locks. 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 locks -- prevent the requesting function on one processor from doing any work until the lock is freed on another processor. The lock manager enters a loop that keeps testing the lock until it is released on the owning processor. As soon as the lock is free, the lock manager spinning on the requesting processor attempts to obtain the lock for the requesting function. As long as a spin lock (except for shared/exclusive locks and the CPU lock) is held by a function executing on a processor, the ID of that processor is in the lockword. Once the lock is released by the owning function, the lockword is cleared.

  - Shared/exclusive locks--serialize the reading or updating of a global resource. More than one processor can own a shared/exclusive lock as shared at one time; only one processor can own a shared/exclusive lock as exclusive at one time.

    Code executing under a shared/exclusive lock is physically disabled. Figure 8 summarizes the results of an unconditional request for a shared/exclusive lock that another processor holds. In general, the lock manager gives processors spinning for exclusive ownership of a shared/exclusive lock priority over processors spinning for shared ownership.

**Note:** The contents of the lockword for a shared/exclusive lock is different from the contents of a spin lockword. In particular, the shared/exclusive lockword does not contain a logical processor ID. For more information about the contents of the lockword for a shared/exclusive lock, see *Diagnostic Techniques.*

| Type of Request | How Held by Owning Processor | Results |
|---|---|---|
| Shared | Shared | Obtain shared ownership. |
| Shared | Exclusive | Spin on the lock until the exclusive owner releases it. |
| Exclusive | Shared | Spin on lock until all shared owners release it.<br>Set the exclusive-pending-request bit in the lockword. |
| Exclusive | Exclusive | Spin on lock until the exclusive owner releases it.<br>Set the exclusive-pending-request bit in the lockword. |

Figure 8. Requests for Shared/Exclusive Locks

- CPU lock--provides system recognized (legal) disablement for units of work (requestors) on a processor level. System recognized (legal) disablement is defined as holding a spin lock or having a super bit set in the PSASUPER field of the PSA. While a requestor holds the CPU lock, the requestor is physically disabled for I/O and external interruptions.

  Multiple units of work on the same processor can own the CPU lock. The CPU lockword (in the PSA) 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.

  **Note:** The CPU lockword does not contain a processor ID. See *Diagnostic Techniques* for additional details about the CPU lockword; see Figure 7 for a description of the "hierarchy" of the CPU lock and its other attributes.

- 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.

## Examples of Lock Types

All of the locks described in Figure 7 with the exception of the CPU, LOCAL, cross memory local (CML), and cross memory services (CMS) locks, are spin locks. The CPU lock can be considered a pseudo spin lock. The LOCAL, CML, CMS, CMSSMF, and CMSEQDQ 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. 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. See *Diagnostic Techniques* for a description of the contents of a local suspend lockword.

**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.

The CMS lock is an enabled global lock for the following reasons:

- Because disabled page faults are not allowed in the system, some global functions need a lock that does not require the functions to fix all their code and control blocks.
- 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 other global locks are disabled spin locks because the functions that run under the locks 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. Also, the more frequently used functions (for example, IOS interruption handler, dispatcher, and storage manager) perform interruption handling and task switching, and have to remain disabled.

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 instruction.)

## Classes of Locks

There are two classes of locks:

- Single locks -- Only one lock exists at a given level of the locking hierarchy. Because there is one lock at a given level, SETLOCK requests for single locks cannot specify the ADDR keyword parameter.
- Multiple locks (commonly referred to as class locks) -- More than one lockword exists at a given level of the locking hierarchy. Because of this, SETLOCK requests for multiple locks must specify the ADDR keyword parameter.

The locks provided in MVS/XA in hierarchical order are:

- RSMGL (real storage management global lock) -- serializes access to all RSM global queues and resources.
- VSMFIX (virtual storage management lock) -- serializes the common area subpools (subpools 226, 227, 228, 231, 239, 241, and 245).
- ASM (auxiliary storage management lock) -- serializes ASM resources on an address space level.
- ASMGL (auxiliary storage management global lock) -- serializes ASM resources on a global level.
- RSMST (real storage management steal lock) -- serializes RSM control blocks on an address space level when it is not known which address space locks are currently held.
- RSMCM (real storage management common lock) -- serializes RSM resources in the common area (such as page table entries, the pageable-frame queue, and the fixed-frame queue).
- RSMXM (real storage management cross memory lock) -- serializes RSM control blocks on an address space level when serialization to a second address space is necessary.
- RSMAD (real storage management address space lock) -- serializes RSM control blocks on an address space level.
- RSM (real storage management lock) -- serializes RSM execution and RSM resources.

- VSMPAG (virtual storage management lock) -- serializes the use of common VSM work area for pageable subpools.
- DISP (global dispatcher lock) -- serializes the use of resources such as address space vector table (ASVT) updating and changes to the address space control block (ASCB) dispatching queue.
- SALLOC (space allocation lock) -- serializes the external receiving routines that enable a processor for either an emergency signal or a malfunction alert.
- IOSYNCH (IOS synchronization lock) -- serializes global IOS functions by means of an IOSYNCH lock table.
- IOSUCB (IOS unit control block lock) -- serializes access and updates to the unit control blocks (UCB)s. There is one lock for each UCB.
- SRM (system resources manager lock) -- serializes use of the SRM control algorithms and associated data.
- TRACE (TRACE lock) -- serializes the system trace buffer structure.
- CPU (processor lock) -- serializes on the processor level, providing system recognized (legal) 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.
- CMSEQDQ (ENQ/DEQ cross memory services lock) -- serializes ENQ/DEQ functions and the use of ENQ/DEQ control blocks.
- CMSSMF (SMF cross memory services lock) -- serializes SMF functions and the use of SMF control blocks.
- 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, CMSEQDQ, or CMSSMF lock. You cannot release the local lock while holding a cross memory services lock. You need not hold all locks in the hierarchy up to the highest lock needed. Hold only locks that you need.

## Locking Hierarchy

The locks are arranged in a hierarchy to prevent a deadlock between functions on the processor(s). An example of a deadlock between functions would be:

- Function A holding the SRM lock and requesting the DISP lock on processor 0
- Function B holding the DISP lock on processor 1 and requesting the SRM lock currently held on processor 0

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 Figure 7, with RSMGL being the highest lock.

The CPU lock has no hierarchical relationship with other spin locks. The CPU lock can be obtained while other spin locks are held; other spin locks can be obtained (in their hierarchical sequence) while the CPU lock is held. The CPU lock is, however, higher in hierarchical order than any of the suspend locks, therefore once you obtain the CPU lock, you cannot obtain any suspend lock. The cross memory services locks (CMS, CMSEQDQ, and CMSSMF) are equal

to each other in the hierarchy. The CML and LOCAL locks are also equal to each other in the hierarchy.

With the exception of cross memory services locks, a processor can hold only one lock at the same level of hierarchy. Therefore, if a processor holds an IOSUCB lock, it may not request a different IOSUCB lock at a different address. If a processor holds one cross memory services lock, it can not request another cross memory services lock. However, a processor can hold all cross memory services locks if it unconditionally requests them simultaneously. If the locks are requested at the same time, they must be released at the same time. It is not recommended that all cross memory services locks be held at the same time because it will degrade performance.

### Using the Same Lockword for Class Locks at Different Levels

To simplify lockword management, a user can provide the same lockword for certain class locks at different levels of the locking hierarchy (for example, the RSMST, RSMCM, RSMXM, and RSMAD locks). However, the lockword can only represent one lock at any given time.

For example, the RSMXM lock held at location 1000 on processor 0 creates two kinds of locking restrictions:

- No other lock (for example, RSMAD) can be obtained at location 1000 on processor 0 or any other processor, until the RSMXM lock is released (however, another lock, like the RSMAD lock, can be obtained at another location on processor 0).
- An RSMXM lock at another location on cannot be obtained on processor 0 until the RSMXM lock at location 1000 is released (however, the RSMXM lock at another location on another processor can be obtained).

The lock manager prevents an interlock by detecting the attempt to simultaneously obtain multiple locks using the same lockword or lock location.

For conditional requests using the same lockword, the lock manager supplies return codes that the user can check. For unconditional requests, if the caller holds the lockword for a different level lock, the lock manager abnormally terminates the caller with an 073 ABEND code. The return codes are described with the syntax of the SETLOCK macro instruction in Volume 2.

There is another situation in which an interlock could occur. This type of interlock is not prevented by the lock manager, but must be solved in the program by using internal hierarchy rules. It involves using the same lockword for a class of locks. For example, if task A and task B are executing on different processors an interlock could occur if:

> Task A holds the RSMAD lock located at location 1000 and requests the RSMXM lock located at location 2000 while task B holds the RSMAD lock located at location 2000 and requests the RSMXM lock located at location 1000.

## 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 one or all of the cross memory services locks (CMS, CMSEQDQ, or CMSSMF).

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 instruction to obtain, release, or test a specified lock or set of locks (using the OBTAIN, RELEASE, and TEST parameters). Users can also obtain the current CPU lock use count for a processor and determine whether a processor holds a spin lock higher in the locking hierarchy than a specified lock. To use SETLOCK, you must be executing in supervisor state with protection key 0. Users of SETLOCK can also be executing in SRB mode, in cross memory mode, as an extension of the interrupt handlers, or as a system service such as the MVS/XA dispatcher.

### Disabled/Enabled State for Obtain

When a global spin type 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.

For an unsuccessful conditional request of a spin lock, control returns to the caller disabled only if the caller was disabled on entry. Otherwise, control returns 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 a global spin type lock is released, control returns to the caller enabled for I/O and external interruptions unless at least one of the following is true:

- Another global spin lock is held
- A disabled supervisor indicator (PSASUPER) is on
- The DISABLED parameter was specified

If one of the above is true, control returns to the caller disabled for I/O and external interruptions.

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.

For a release request via the SPIN, ALL, or (reg) parameters, the final state is the same as that which would have existed had the locks been released one at a time.

### Altering the Dispatching Queue (INTSECT)

The intersect function is the serialization mechanism that the dispatcher and control program functions use to alter the dispatching queues. The LOCAL and dispatcher locks are used in conjunction with the intersect function. Intersect serialization is only between the requestor of the intersect and the dispatcher. The requesting routine must hold the LOCAL or dispatcher lock for serialization with other routines.

A routine can intersect on either the local or global level. The LOCAL lock is required for obtaining the local intersect; it also ensures the proper serialization with other routines requesting the local intersect. The local intersect ensures serialization of an address space with the dispatcher and serialization of routines that modify the TCB dispatching queue or TCB dispatchability. Similarly, the dispatcher lock is required for routines requesting the global intersect. The global intersect ensures serialization of dispatcher functions on a global level.

## Using the Must-Complete Function (ENQ/DEQ)

System routines (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 nondispatchable state until the requesting task -- the task issuing an ENQ macro instruction 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 instruction 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 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 problem program task in an address space is allowed to execute. All other problem program tasks, and the initiator task, 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. The initiator and all other tasks in the job step are set nondispatchable. 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 instruction 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. The macro instructions ATTACH, LINK, LOAD, and XCTL 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 macro instructions either has been requested by one of the tasks made nondispatchable by the use of the SMC parameter or was requested by another task and has been only partially fetched.
5. 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 instruction to terminate the must complete mode as soon as processing is finished. Tasks set nondispatchable by the corresponding ENQ macro instruction are made dispatchable and asynchronous exits from the requesting task are enabled.

# Limiting Global Resource Serialization Requests

Global resource serialization allows an installation to share symbolically-referenced resources between units of work. (*Planning: Global Resource Serialization* explains the function and use of global resource serialization.) A global resource serialization request is an ENQ or RESERVE request that causes an element to be added to any queue in the global resource serialization queue area.

GQSCAN uses the same control blocks as ENQ and RESERVE to obtain the status of resources and requestors of resources during resumption processing. (The GQSCAN macro is described in the section 'Measuring System Characteristics' and in Volume 2.) In order to prevent any one job, started task, or TSO user from generating too many concurrent requests, global resource serialization limits the number of global resource serialization requests in each address space.

Global resource serialization counts the number of ENQ/RESERVE requests and the number of pending GQSCAN requests issued by all TCBs in each address space. Each time a user issues an ENQ/RESERVE, global resource serialization increases the count in that address space by 1 for each resource name and decreases the count by 1 when a user in that address space issues a DEQ. Similarly, when a user issues a GQSCAN request, global resource serialization increases the count in that address space by 1 and decreases the count by 1 when the scan is completed (if resumption is requested).

Global resource serialization compares the computed count of requests to a threshold value (4096) stored in the GVTCREQ field of the global resource serialization vector table (GVT). (See *SPL: System Modifications* for a description of how to change the threshold value.) When the computed count reaches the threshold value, global resource serialization processes subsequent requests as follows:

- ENQ/RESERVE requests from unauthorized callers are rejected; unconditional requests from these callers are abended and conditional requests receive a return code of X'18'.
- ENQ/RESERVE requests from authorized callers are not rejected until the count exceeds the threshold value by a tolerance value. This higher limit is stored in the GVTCREQA field of the GVT. The tolerance provided by the system is 15, but system programmers can change this value. (See *SPL: System Modifications* for a description of how to change the limit for authorized callers.) This means that an additional 15 concurrent ENQ/RESERVE requests are accepted from authorized callers. This is done to allow for normal termination and to permit error recovery routines to obtain the resources that they need. Once the computed count exceeds the limit for authorized callers, subsequent requests are rejected in the same way as requests from unauthorized callers.
- GQSCAN requests that do not fit into the caller's buffer receive a return code of X'14'; these requests are not queued and a TOKEN is not provided.

ENQ/RESERVE requests from authorized callers use the MASID and MTCB parameters to allow a further conditional control of a resource. One task issues an ENQ or RESERVE for a resource specifying a matching ASID; if the issuing task does not receive control, it is notified whether the matching task has control (which allows the issuing task to use the resource even though it could not acquire the resource itself). This process requires serialization between the issuing and requesting tasks.

# 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, 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. This topic, as it relates to macro instructions, is discussed further under "Macro Instructions Used with Shared DASD." 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."

## Devices that Can be Shared

The following control units and devices are supported by the shared DASD option:

- IBM 2835 Storage Control Unit with two-channel switch -- IBM 2305 Fixed Head Storage Facility.
- IBM 3830 Storage Control Unit with two-channel switch -- IBM 3330 Series Disk Storage Drive. The IBM 3330, 3340/3344, 3350 Series devices may also be configured for shared use via the string switch feature.
- IBM 3880 Storage Control Units Models 2 and 3 -- IBM 3380 Direct Access Storage Facility.

- IBM 3880 Storage Control Units Models 1 and 2 -- IBM 3375 Direct Access Storage Facility.

Alternate channels to a device from any one system can only be specified for the IBM 3330, 3340/3344, 3350 Series Storage Unit, the 3375 Direct Access Storage Device, and the 3380 Direct Access Storage Device.

## Volume/Device Status

The shared DASD facility requires that certain combinations of volume characteristics and device status be in effect for shared volumes of devices. Figure 9 shows the combinations that must be in effect for a volume or device:

| 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.) |

Figure 9. Valid Volume Characteristic and Device Status Combinations

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.

## System Configuration

Operating system configurations do not have to be identical to share a data set. The only additional equipment needed for the Shared DASD option is either a two-channel switch or a 2844 Auxiliary Control unit. The user must also observe certain restrictions about the data sets that are shared. The following data sets cannot be shared:

```
Master catalog
PASSWORD          SYS1.MANx
SYS1.DCMLIB       SYS1.NUCLEUS
SYS1.DUMPxx       SYS1.PAGExx
SYS1.LOGREC       SYS1.STGINDEX
SYS1.LPALIB       SYS1.SVCLIB
```

Because the system does not provide for the sharing of the PASSWORD data set, the PASSWORD data set for each system must contain password records for all protected data sets. Where independent computing systems share common DASD resources, individual installations must ensure that the PASSWORD data set contains records for all protected data sets for each system sharing the DASD. For further details regarding password protection on shared DASD, see *System-Data Administration.*

## 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 dismounted.
- 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 shareable devices offline before issuing START commands and then following parallel mount procedures.

## Macro Instructions Used with Shared DASD (RESERVE, EXTRACT)

You can use the RESERVE, ENQ, DEQ, and EXTRACT macro instructions when working with shared DASD. The following paragraphs describe the use of these macro instructions in relation to shared DASD.

- The RESERVE macro instruction reserves a device (identified by its UCB address and symbolic resource name) for use by a particular system. Each task that needs exclusive use of a device must issue the RESERVE macro. When a task issues a RESERVE for a particular device, the system increases the count of outstanding reserve requests (located in the UCBSQC field of the UCB) for that device. If MVS starts I/O for that device while the count is non-zero, it precedes the channel program with a RESERVE channel control word (CCW) that reserves the device to the system that executed the RESERVE CCW.

  **Notes:**

  1. The set-must-complete (SMC) parameter of the ENQ macro instruction can also be used with RESERVE.
  2. If a check point 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 initiator, allocation, and direct access device storage management (DADSM) components of MVS use an ENQ with a major name of SYSDSN to serialize access to datasets, whether these datasets exist on shared DASD or not. The use of global resource serialization in a shared DASD environment allows DADSM to serialize the use of DASD space by all systems sharing a volume so that data integrity is guaranteed. The SYSDSN major name must be a resource known to all systems in the shared DASD environment. Refer to the topic Setting Up Resources Name Lists in the "Global Resource Serialization" section for more information about how to make SYSDSN known to all systems.
- 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 starts a channel program, consisting of a RELEASE CCW, to free the device.
- If global resource serialization is active, ENQ and DEQ, with SCOPE=SYSTEMS specified, can serialize on a particular shared DASD data set without reserving the entire device. See *Planning: Global Resource Serialization* for details.
- The EXTRACT macro instruction obtains 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" explains this procedure. EXTRACT provides information, it does not actually serialize a resource.

## Releasing Devices

The DEQ macro instruction 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=pointer address parameter is not required. If the DEQ macro instruction is not issued by a task that has previously reserved a device, the system frees the device when the task is terminated.

## 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 instruction 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 ENQ to serialize the resource. See *Planning: Global Resource Serialization* for additional information on this topic.

## 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 10, 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.

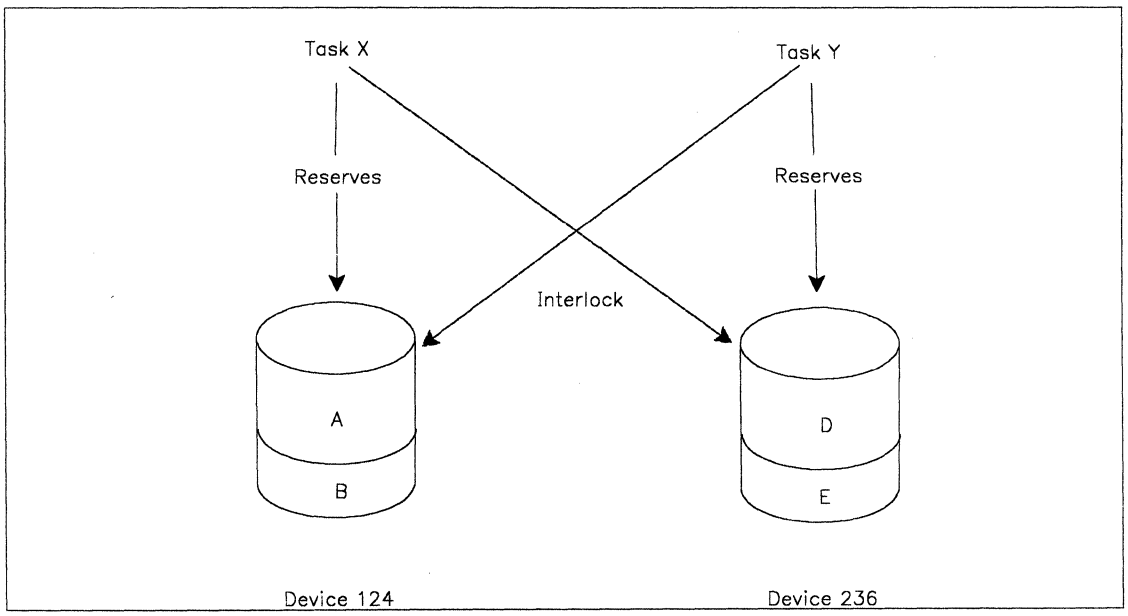



Figure 10. Example of an Interlock Environment

### 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. Since the SYS1.LINKLIB for each system resides on a volume reserved by the other system, the XCTL macro instruction 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 *Planning: Global Resource Serialization* for additional information about RNLs and about preventing interlocks.

### Finding the UCB Address for the RESERVE Macro

This topic explains procedures for finding the UCB address for use by the RESERVE macro instruction; it also shows a sample assembler language subroutine that issues the RESERVE and DEQ macro instructions and can be called by routines written in higher level languages.

Job management routines construct the TIOT, which resides in virtual storage during step execution. The TIOT consists of one or more DD entries, each of which represents a data set defined by a DD statement for the jobstep. Each entry includes the DD name. Associated with each DD entry is the UCB address of the associated device. In order to find the UCB address, you must locate the DD entry in the TIOT corresponding to the DD name of the data set for which the RESERVE macro instruction is to be issued.

**Providing the Unit Control Block Address to RESERVE:** Use the EXTRACT macro instruction 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 instruction. Before issuing an EXTRACT macro instruction, set up an answer area to receive the requested information. One full word 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 instruction.

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

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, offset 44 decimal 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 instruction. (In disk addresses of the form MBBCCHHR, the M indicates the extent number starting with 0).

**Procedures for Finding the UCB Address of a Device:**

- For data sets using the queued access methods in the update mode or for unopened data sets:
  1. Extract the TIOT from the TCB.
  2. Search the TIOT for the DD name associated with the shared data set.
  3. Add 16 to the address of the DD entry found in step 2. This results in a pointer to the UCB address in the TIOT.
  4. Issue the RESERVE macro specifying the address obtained in step 3 as the parameter of the UCB keyword.

  **Note:** This procedure can be used for non-concatenated DD statements and for data sets that reside on a single volume.
- For opened data sets:
  1. Load the DEB address from the DCB field labeled DCBDEBAD.
  2. Load the address of the field labeled DEBDVMOD in the DEB obtained in step 1. The result is a pointer to the UCB address in the DEB.
  3. Issue the RESERVE macro specifying the address obtained in step 2 as the parameter of the UCB keyword.
- 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 DCBDEBAD.
  4. Load the address of the field labeled DEBDVMOD in the DEB.
  5. Multiply the "M" of the actual device address (step 2) by 16.
  6. 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.
  7. Issue the RESERVE macro specifying the address obtained in step 6 as the parameter of the UCB keyword.

- If the data set is an ISAM data set, QISAM in the load mode should be used only at system update time. Further, if it is a multivolume ISAM data set, it must be assumed that all jobs will access the data set through the highest level index. The indexes should never reside in virtual storage when the data set is being shared. In this case, by issuing a RESERVE macro for the volume on which the highest level index resides, the user effectively reserves the volumes on which the prime data and independent overflow areas reside. The following procedures can be used to achieve this:

  1. Open the data set.
  2. Locate the actual device address (MBBCCHHR) of the highest level index. This address can be obtained from the DCB.
  3. Load the DEB address from the DCB field labeled DCBDEBAD.
  4. Load the address of the field labeled DEBDVMOD in the DEB.
  5. Multiply the "M" of the actual device address located in step 2 by 16.
  6. 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.
  7. 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 *VSAM Administration Guide*.

**RESDEQ Subroutine:** The assembler language subroutine in Figure 11 can be used by assembler language programs to issue the RESERVE and DEQ macro instructions. Parameters that must be passed to the RESDEQ routine, if the RESERVE macro instruction 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 instruction 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
                                    ZEROS
         CLC     0(4,3),=F'0'       WORD OF ZEROS IF DEQ IS
                                    REQUESTED
         BE      WANTDEQ
*PROCESS FOR DETERMINING THE UCB ADDRESS USING THE TIOT
         XR      11,11              REGISTER USED FOR DD ENTRY
         EXTRACT ADDRTIOT,FIELDS=TIOT
         L       7,ADDRTIOT         ADDRESS OF TASK I/O TABLE
         LA      7,24(7)            ADDRESS OF FIRST DD ENTRY
NEXTDD   CLC     0(8,3),4(7)        COMPARE DDNAMES
         BE      FINDUCB
         IC      11,0(7)            LENGTH OF DD ENTRY
         LA      7,0(7,11)          ADDRESS OF NEXT DD ENTRY
         CLC     0(4,7),=F'0'     CHECK FOR END OF TIOT
         BNE     NEXTDD
         ABEND   200,DUMP           DDNAME IS NOT IN TIOT, ERROR
FINDUCB  LA      8,16(7)            ADDRESS OF WORD IN TIOT THAT
*                                   CONTAINS ADDRESS OF UCB
*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=(8)
         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)
ADDRTIOT DC      F'0'
SAVE     DS      18F
QNAME    DS      2F
RNAME    DS      CL256
RNLEN    DC      F'0'
         END
```

Figure 11. Example of Subroutine Issuing RESERVE and DEQ

*Note*: This example assumes that non-concatenated DD statements and single volume data sets are used.

# Indicating Event Completion (POST)

The POST macro instruction 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.

## Cross Memory POST

The authorized user (executing in supervisor state, under protection key 0-7, or APF-authorized) of the POST macro instruction can use the ASCB and ERRET parameters to schedule an SRB to be dispatched to perform a POST in an address space other than his 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 instruction is sometimes known as "cross memory post."

The ERRET routine is given control in the issuer's address space when an error condition is detected. It receives control enabled, unlocked, in SRB mode, and with the following register contents:

| Register | Contents |
|---|---|
| 0 | ECB address |
| 1 | address of POSTer's ASCB |
| 2 | completion code specified on POST invocation |
| 3 | completion code from failing address space |
| 4-13 | unpredictable |
| 14 | return address |
| 15 | ERRET address |

The ERRET routine will receive control in the addressing mode of the caller of the cross memory POST. 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 instruction. The ECB might or might not be posted, depending on existing conditions. Because SPOST invokes the PURGEDQ SVC, see the description of PURGEDQ for the restrictions on its use.

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

The programmer 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 12 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 12. Bypassing the POST Routine

## Waiting for Event Completion (EVENTS)

The EVENTS macro instruction 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, significantly more efficient than SVC entry, is available to users executing in key 0, supervisor state, and holding only the LOCAL lock.

Branch entry specifies BRANCH=YES on the EVENTS macro instruction. 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 dispatcher, even though there might be ECBs posted to the EVENTS table. EVENTS frees the LOCAL lock. Before issuing the EVENTS macro instruction with the WAIT=YES option, you must establish the return environment (the PSW and registers in the RB and TCB). EVENTS stores a pointer to the first completed EVENTS entry into the TCB register 1 save location. (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 system routines with a service that allows them to receive control immediately upon each completion of an outstanding event. 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

IEA0PT0E is the entry point to POST. It performs exit identification and deletion 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 Macro Instruction."

You cannot provide the same exit routine as input to IEA0PT0E on separate invocations in different addressing modes. 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 instruction. 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. It has the format shown in Figure 13.

| VALUE (1 byte) | MODE (1 byte) | RESERVED (2 bytes) |
|---|---|---|
| POST DATA (4 bytes) | | |

Figure 13. ECB Extension (ECBE)

The fields in the ECBE are:

**VALUE** is one byte containing a value from 1-255. A value of 1 indicates that the POST exit function is being requested. All other function codes are reserved.

**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. The first bit of this byte must match the addressing mode that existed when IEA0PT0E was invoked to identify the exit routine.

**POST DATA** When VALUE is 1 (that is, contains X'01') 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 to the format shown in Figure 14. 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 an extended ECB. The meaning of the bits in the extended ECB follows:

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

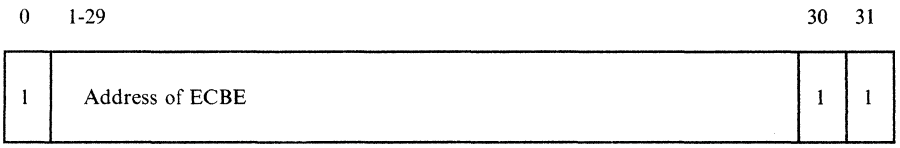



Figure 14. Extended ECB

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.

Once the extended ECB has been initialized, the LOCAL lock must be held when modifying the ECB. If the LOCAL lock is not held, posts may be lost.

## 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.

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 |

The register contents upon return to POST from the user's routine must be:

Register 11, 14-Unchanged
Register 0-10, 12, 13, 15-Irrelevant

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 available when the routine receives control).

## Re-entry to POST from a POST Exit

A POST exit routine can issue POST only via the POST entry point, IEA0PT03. Details of the interface are in "Branch Entry to the POST Service Routine."

Because of the save area recursion within POST, 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 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.

## Example of Using a POST Exit Function

A subsystem allocates and initializes extended ECBs, ECBEs, and EQTs. These data areas appear in Figure 15. Once initialized, the subsystem dispatcher waits on a list of ECBs. Each list entry identifies an ECB in an EQT.

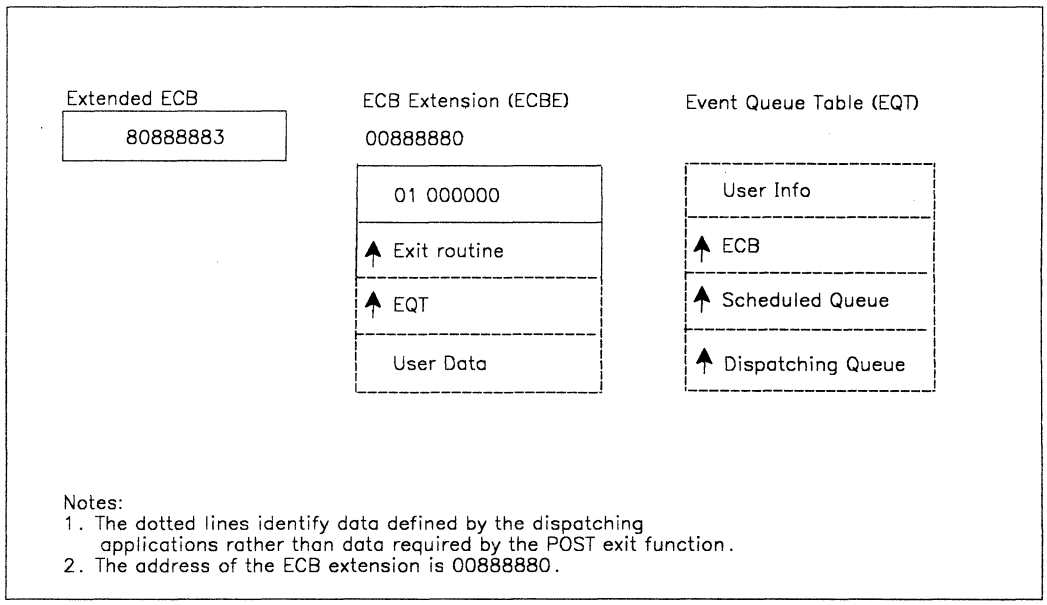



Figure 15. Data Areas Post Exit Example

As soon as any system routine posts an extended ECB, the subsystem exit routine identified in the ECBE receives control. The exit routine receives control in the addressing mode specified by the first bit in byte 1 of the ECBE. If this bit is on, the exit routine receives control in 31-bit addressing mode; if this bit is off, the exit routine receives control in 24-bit addressing mode. After checking the validity of the work request, the exit routine places a work element on the EQT schedule queue identified by the ECBE. The exit routine then posts the ECB associated with that EQT, which completes the queuing of the work and the activation of the dispatching task.

The subsystem dispatcher then scans the ECB list to locate posted ECBs (that is, an EQT with work queued). The subsystem dispatcher then uses compare and swap to switch the schedule queue to the dispatching queue values. Then the subsystem dispatcher dequeues work from the dispatching queue until the queue is depleted. Then the subsystem dispatcher clears the post bit in that EQT ECB and again uses compare and swap to move the schedule queue to the dispatching queue. If the dispatching queue is still empty, the subsystem dispatcher checks the next ECB in the ECB list. After having processed the entire ECB list, the subsystem dispatcher again awaits requests for work.

The subsystem dispatcher can use the USERINFO field in the EQT for serialization where multiple system dispatcher tasks can wait on the same EQT.

## Branch Entry to the POST Service Routine

Branch entry to the POST service routine provides all the normal ECB and RB POST processing. The BRANCH parameter on the POST macro instruction uses entry point IEA0PT01 only. To use the other entry points, shown in Figure 16, 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.

Figure 16 shows the POST function and the branch entry points through which those functions can be performed. Figure 17 shows the input parameters to POST. Figure 18 shows the output parameters from POST.

| **Functions** | **Entry Points** | | | |
|---|---|---|---|---|
| | IEA0PT01 (CVT0PT01) | IEA0PT02 (CVT0PT02) | IEA0PT03* (CVT0PT03) | IEA0PT0E (CVT0PT0E) |
| Local ECB POST | X | X | X | |
| Local POST without ECB | X | | X | |
| Cross address space POST | X** | | X | |
| Post exit creation/deletion | | | | X |

* This entry point performs processing identical to entry point IEA0PT01. 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 IEA0PT0E).

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

Figure 16. POST Function and Branch Entry Points

| Registers | IEA0PT01 (CVT0PT01) | IEA0PT02 (CVT0PT02) | IEA0PT03 (CVT0PT03) | IEA0PT0E (CVT0PT0E) |
|---|---|---|---|---|
| 0 | ECB storage protect key[1] | | ECB storage protect key[1] | Func. Code |
| 1 | | | | Exit Routine Address |
| 10 | Completion Code[2] | Completion Code | Completion Code[2] | |
| 11 | ECB Address[3] | ECB Address | ECB Address[3] | |
| 12 | Error Routine Address[4] | | Error Routine Address[4] | |
| 13 | ASCB Address[4] | | ASCB Address[4] | |
| 14 | Return Address | Return Address | Return Address | Return Address |
| 15 | Entry Point Address | Entry Point Address | Entry Point Address | Entry Point Address |

[1] If cross address space post, optionally contains the storage protection key of the ECB in bits 24-27.

[2] If POST-without-ECB, contains RB address; if cross address space 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 POST-without-ECB, set to zero; if local address space POST, ensure high-order bit of register is zero; if cross address space POST, set high-order bit of register to 1.

[4] Only necessary when performing cross address space POST. If performing a cross address space 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.

Figure 17. POST Branch Entry Input

| Entry Points | Registers Saved and Restored |
|---|---|
| IEA0PT01[1] | 0-9, 12[2], 13[2], 14 |
| IEA0PT02 | 0-9, 12-14 |
| IEA0PT03 | 0-14 |
| IEA0PT0E | 2-14 |

[1] The contents of only registers 9 and 14 are retained during a cross address space POST when either the LOCAL lock is not held or the high order bit in register 12 is on; all other register contents are unpredictable.

[2] The contents of these registers will not be saved and restored during a cross address space POST; their contents are therefore unpredictable in these circumstances.

Figure 18. POST Branch Entry Output

## Branch Entry to the WAIT Service Routine

Branch entry to the WAIT service routine provides all the normal ECB and RB WAIT processing. This function is not available, however, to Type 1 SVCs or SRBs. The caller must hold home's LOCAL lock and be in key zero, supervisor state with current addressability to the home address space. 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. 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, (user data cannot be passed through this field or register).
- Releases home's LOCAL lock.
- Returns control to the dispatcher (control does not return to the caller even though all previously pending events have already occurred). The dispatcher ensures that all FRRs have been deleted.

Branch entry to WAIT can occur without identification of any ECBs. This process sets the wait count in the current RB to the specified value. The corresponding POSTs-without-ECB then activate the RB. If you use this process, make sure that the WAIT-without-ECB precedes the POST-without-ECB in order to avoid causing the RB to wait indefinitely.

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. If the WAIT-without-ECB function is being requested, set register 1 to a value of zero. |
| 15 | The branch entry address to WAIT (IEAVWAIT), which in turn is obtained from the CVT (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.

## Suspension and Resumption of Request Blocks

An alternate method of waiting for an event and indicating its completion is available on a restricted basis for systems programming. This method gives faster performance than the normal method of using the WAIT and POST macro instructions. The summary below outlines the functions that provide this alternative:

| Macro | Description |
|---|---|
| SUSPEND | Wait for an event to complete. |
| RESUME | Indicate the completion of the event. |
| TCTL | Give control directly to a ready task. |
| CALLDISP | Give up control so that an event can complete. |

## Waiting for an Event to Complete (SUSPEND)

The SUSPEND macro instruction provides an efficient means of waiting for an event to complete. It is analogous to the WAIT macro instruction, and is used in a SUSPEND-RESUME sequence, which is analogous to the WAIT-POST sequence. The SUSPEND macro instruction causes the wait for event completion through the wait count field (RBWCF) in the request block (RB). This field is the same one the WAIT macro instruction uses. When used with the SUSPEND macro instruction, however, the wait count field is known as the suspend count field, even though the function it performs for both macro instructions is the same.

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

RBs that issue the SUSPEND macro instruction 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 exit to the dispatcher (for example, issue a CALLDISP macro instruction with the BRANCH=YES option). Using the SUSPEND macro instruction 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 macro instructions.

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 instruction sequences:

### Scenario 1:

SUSPEND RB=PREVIOUS

1. Type 2 SVC routine receives control.
2. The SVC suspends 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 issuer of the macro instruction.
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. Macro issuer issues CALLDISP BRANCH=YES, which releases the LOCAL lock or CML lock before going to the dispatcher.

5. Event completion occurs; process started in step 3 resumes issuer of the macro instruction.
6. Normal processing resumes.

Consider the following when using the SUSPEND macro instruction:

- The SUSPEND macro instruction can be issued in cross memory mode.
- Only a routine executing under protection key 0 can issue SUSPEND.
- The SUSPEND macro instruction requires that the CVT mapping macro be included.
- 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.
- Only task-related users can issue SUSPEND, and then only for the current task. SUSPEND cannot be issued for another TCB or by an SRB.
- 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.
- 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.
- When using the SUSPEND RB=CURRENT option, the issuer must either execute disabled or hold the 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 the 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 use issue CALLDISP FRRSTK=SAVE to enter the dispatcher. 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 instruction or release home's LOCAL lock, and it must exit via its exit mechanism. A Type 6 SVC must not issue the CALLDISP macro instruction 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 an entry to the dispatcher (via CALLDISP) 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).
- The SUSPEND and RESUME sequence must not be intermixed with the WAIT and POST sequence on a single RB because both sequences use the same count field for control of the functions. 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, the count would be reset to one.
- 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.

- A program that has invoked the SUSPEND RB=CURRENT option must not be suspended again (for example, through a page fault or a lock suspension) after releasing home's LOCAL lock, a CML lock, or enabling until a RESUME is issued counteracting the outstanding SUSPEND macro instruction.

## Resuming Execution of a Suspended Request Block (RESUME)

The RESUME macro instruction, which is supported in cross memory mode, provides an efficient means for indicating the completion of an event. The RESUME macro instruction specifies the TCB and RB that were previously suspended by the SUSPEND macro instruction. 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 instruction.

The RESUME macro instruction 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 instruction has the MODE and ASYNC options to handle these types of situations.

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

The MODE option specifies whether or not the RESUME operation must complete (MODE=UNCOND) or not (MODE=COND). MODE=UNCOND requires that certain system locks can be obtained.

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 disabled and holding 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 instruction, and then determined that there was more work to be done. The disabled, unlocked routine could issue a RESUME macro instruction 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 instruction with the MODE=UNCOND and ASYNC=N options.
  - The RETURN=N option on the RESUME macro instruction 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 home addressing mode, and must not hold any locks. If these three conditions are met, the TCTL service is entered to transfer 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 obtains an SRB from the supervisor SRB pool and schedules it to complete the RESUME asynchronously.
  - The caller can be enabled or disabled, however, the VSMFIX lock must be available. The caller must not hold any locks higher in the lock hierarchy than the VSMFIX lock or the caller must hold the VSMFIX lock when the RESUME macro instruction is issued.
- 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 can not 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 conditionally obtains an SRB from the supervisor SRB pool and schedules it to perform the RESUME asynchronously. If the supervisor SRB pool is empty, RESUME returns to the caller without completing the RESUME operation.
  - 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.

RESUME provides return codes in register 15 to indicate the result of the RESUME attempt. See the RESUME macro instruction in Volume 2 for details on the return codes.

The RESUME macro instruction requires the IHAPSA mapping macro. If the ASCB option is not specified, then the MODE=UNCOND and ASYNC=Y combination requires the CVT mapping macro. All other combinations require the IHASVT mapping macro.

## Transferring Control for SRB Processing (TCTL)

The TCTL (transfer control) macro instruction allows an SRB routine to exit from its processing and to pass control to a task with minimal dispatcher 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 it passes all the dispatchability tests the dispatcher normally makes.

Some other considerations for using the TCTL macro instruction are:

- The TCTL macro may 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 constitutes an exit from the issuing routine, which therefore causes cleanup of the SRB.
- The TCTL service requires inclusion of the CVT mapping macro.

The TCTL service requires that the SRB requesting the TCTL must not hold any locks and must be in home addressing mode.

## Using the BRANCH = YES Option of CALLDISP (CALLDISP)

The CALLDISP macro instruction with the BRANCH = YES option is supported in cross memory mode. 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 supervisor to save status and control to pass to the dispatcher.

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

- The issuer of CALLDISP must be executing in supervisor state with PSW key zero.
- The issuer must be in task mode rather than in SRB mode.
- The BRANCH = YES option requires inclusion of these mapping macros:

  ```
  IHASVT
  IHAPSA
  ```

- The FIXED = YES or FIXED = NO option can be specified with BRANCH = YES.
- When FRRSTK = SAVE is specified:

  The caller must not hold any locks or an abend results.

  **Note:** For MVS/System Product Version 2 Release 1.3 Vector Facility Enhancement or MVS/System Product Version 2 Release 1.3 Availability Enhancement and later releases:

  - If any EUT (enabled unlocked task) FRRs exist, the current FRR stack is saved and the caller may hold either the LOCAL or CML lock. CALLDISP releases the lock before going to the dispatcher.
  - If no EUT FRR exists, the caller cannot hold any locks. Otherwise, an abend occurs.
- 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 before going to the dispatcher.

**Notes:**

1. A type 1 and type 6 SVC must not issue the CALLDISP macro instruction.
2. The LOCAL or CML lock can be used to serialize the SUSPEND processing and establish the RESUME processing. If a local lock is not used to serialize the FRR stack, the caller can use the CALLDISP FRRSTK = SAVE option to serialize the stack. For more information, see "Suspension and Resumption of Request Blocks" in this volume.

# Reporting System Characteristics

This chapter describes three ways to report system characteristics:

- Using GQSCAN to obtain resource usage reports
- Using SRM to obtain subsystem measurement reports
- Using SYMREC to obtain software error reports

## Collecting Information About Resources and Their Requestors (GQSCAN)

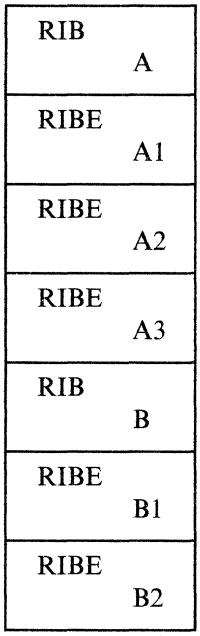
Global resource serialization enables an installation to share symbolically-named resources between units of work. Programs issue ENQ, DEQ, and RESERVE macro instructions to request access to resources. Global resource serialization runs in its own address space and maintains the resource queues in this address space, which cannot be swapped, cancelled, or forced. The only way you can extract information from the resource queues is by using the GQSCAN macro instruction. You can extract this information whether or not global resource serialization is active. See *Planning: Global Resource Serialization* for information about how global resource serialization functions.

Using GQSCAN, you can inquire about a particular scope of resources (such as STEP, SYSTEM, or SYSTEMS), a specific resource by name, a specific system's resources, a specific address space's resources, or resources requested by the RESERVE macro instruction. The GQSCAN service routine collects the information you request from the resource queues and consolidates that information before returning it.

In order to specify a scope of LOCAL or GLOBAL, you must be a supervisor state, key zero user. SCOPE=LOCAL and SCOPE=GLOBAL are restricted because when you specify these parameters the GQSCAN service routine serializes the use of the GRS control blocks to stop any other ENQ, DEQ, or RESERVE instructions from changing them.

The information you request is returned in an area whose location and size you specify using the AREA parameter on the GQSCAN macro instruction. You can also specify a fullword location in which the GQSCAN service routine can return a token by specifying the TOKEN parameter. If the amount of information exceeds the size of your area, the GQSCAN service routine gives you as much as the area will hold and returns a token in the specified location. On subsequent invocations of the GQSCAN macro instruction, if you provide the token, you can obtain the remaining information. You can invoke the GQSCAN macro instruction again and again using the same token, until all of the information has been returned. You can also invoke the GQSCAN macro instruction using the same token and QUIT=YES to terminate the scan.

The information is returned in the form of resource information blocks and resource information block extensions, as shown below.




Resource information block (RIB) describes a resource

RIB extension (RIBE) describes resource requestor

The RIB and RIBE are described in the *Debugging Handbook*.

The amount of information you get about a particular resource depends on the scope you specify on the GQSCAN macro instruction, the size of the area you provide, and whether or not you specify a token.

Whether you specify a scope of STEP, SYSTEM, SYSTEMS, or ALL with or without a token, the information returned the first time you issue the GQSCAN macro instruction is the same. You get the first RIB and as many of its associated RIBEs as will fit in your area. The RIB has an entry that tells how many RIBEs it has and, of those, how many appear in your area. Any RIBEs that do not fit are lost. If there is room for another RIB and all of its RIBEs, it also is returned and so on until there is not enough room for the next RIB and all of its RIBEs.

The second return of information occurs the next time you issue the GQSCAN macro instruction. The contents of this return differ depending on whether or not you specify a token. If you specified a token on the first GQSCAN and supply that token on the next invocation, the information returned continues with the next RIB, and as many of its associated RIBEs as will fit in your area. Any RIBEs that do not fit are lost. If there is room for another RIB and all of its RIBEs, it too is returned, and so on, until there is not enough room for the next RIB and all of its RIBEs. If you do not specify a token, the information starts over with the first RIB.

The example in Figure 19 shows three returns. The scope is either STEP, SYSTEM, SYSTEMS, or ALL and a token is specified each time.

For the example, assume that there are four resources, A, B, C, and D. Resource A has three requestors, resource B has six requestors, resource C has two requestors, and resource D has one requestor.

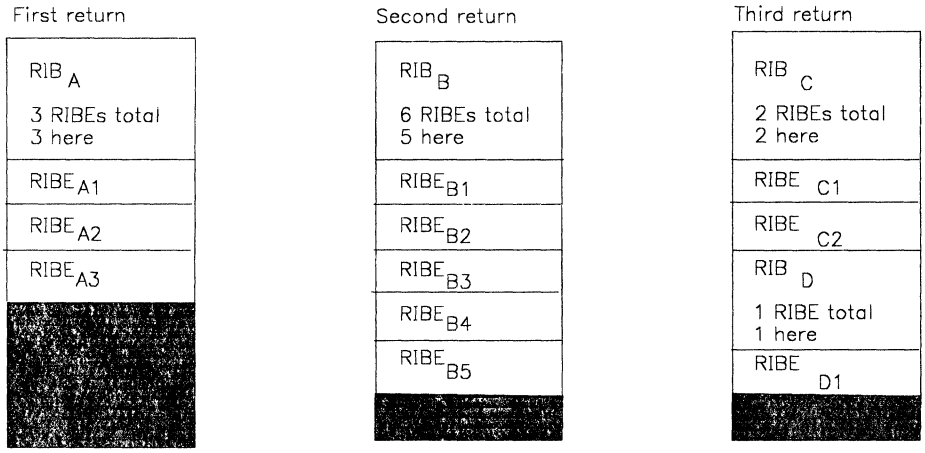



Figure 19. GQSCAN Results with STEP, SYSTEM, SYSTEMS, or ALL

Whether you specify a scope of LOCAL or GLOBAL with or without a token, the information returned the first time you issue the GQSCAN macro instruction is the same. You get the first RIB and as many of its associated RIBEs as will fit in your area. The RIB has an entry that tells the number of RIBEs associated with it, and, of those, how many appear in your area. Any RIBEs that do not fit will appear in your area the next time you issue the GQSCAN macro instruction if a token is provided. If there is room for another RIB and at least one RIBE, it is also returned, and so on , until there is not enough room for a RIB and RIBE combination. If you do not specify a token, the information starts over with the first RIB.

The example in Figure 20 shows two returns. The scope is either LOCAL or GLOBAL and a token is specified each time. Note that the GQSCAN service routine does not truncate any of the RIBEs. On every return after the first, in which you supply the token that was returned, the previous RIB is repeated to put the remaining RIBEs in context and more resource information, if any, continues.

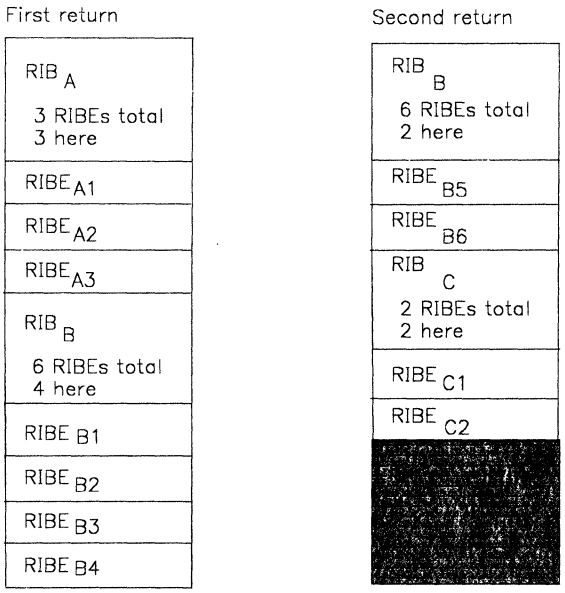



Figure 20. GQSCAN Results with LOCAL or GLOBAL

In scanning the information returned, be sure to 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 is in the high-order half of register 0 and the size of the RIBE is in the low-order half.

The first RIB starts at the beginning of the area you specify. The first RIBE is pointed to by the current RIB pointer plus the size of the fixed portion of RIB plus the size of the variable portion of RIB (RIBVLEN). To find the second RIBE, add the size of the RIBE.

To find the second RIB, use the current RIB pointer plus the size of the fixed portion of the RIB plus RIBVLEN and then add the number of RIBEs times the RIBE size.

## Using the SRM Reporting Interface to Measure Subsystem Activity

The reporting interface allows an IBM or user-written 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, or userid) specified in the installation control specification (IEAICSxx parmlib member). For more information on the installation control specification, see *Initialization and Tuning*.

The reporting interface is necessary because, except for TSO, 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 instruction, which the subsystem must issue at the completion of each transaction. Issuing the macro instruction 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.

# Reporting Software Error Symptoms (SYMREC)

SYMREC is a facility that can store and format information about non-abend errors that occur in an authorized user application. Authorized applications may be written to detect their own errors and to invoke the SYMREC macro, which stores information about each error on SYS1.LOGREC, the on-line repository where error information is collected. The unit of information that the SYMREC macro stores in the repository is called a **symptom record**. The data in the symptom record is a description of some programming failure combined with a description of the environment where the failure occurred.

The SYMREC facility can interpret the symptom records stored on the repository, and formatting them into various kinds of reports. SYMREC consists of the following elements:

1. Two macros, ADSR and SYMREC.
2. Formatting capabilities that are invoked through EREP and IPCS.

## Writing Applications That Use SYMREC

The use of SYMREC implies that an application is written in a programming style in which the program monitors execution-time conditions that denote errors. To write an application that uses SYMREC, you use the ADSR macro to obtain a DSECT for an area in your virtual storage that is called a **symptom record**. This area has six sections. When the application starts to execute, it first stores zeroes into the entire area, then initializes certain fields in the first two sections. These fields identify the application's environment and supply essential addressing information for the application as it executes. After identifying the symptom record, the application continues with its main processing.

If the application detects an error during its processing, it collects information about the error, then stores the error information in the symptom record by referencing the data fields in the ADSR DSECT. When the application stores the error information, the information is in a special format called the **SDB** (structured data base) format.

After storing data into the symptom record, the application issues the SYMREC macro, which writes data from the symptom record to an on line repository. Before SYMREC processing writes the data, it updates the symptom record in the application's virtual storage (even if the storage is store-protected) with various data about the execution environment. After SYMREC writes the data, it returns control to the application with return and reason codes respectively in registers 15 and 0. The application can continue processing if the error is not fatal. The return and reason codes, in hexadecimal, are:

| Return Code | Reason Code | Explanation |
|---|---|---|
| 0000 | | Symptom record component completed successfully and the symptom record was recorded. |
| | 0000 | Successful completion of the SYMREC macro service routine. |
| 0004 | | Error(s) detected on the SYMREC macro statement. The entire input record was recorded. Following are specific reasons why the symptom record component processed unsuccessfully: |
| | 0164 | The input symptom record was successfully copied. However, an attempt to write section 1 information from the completed symptom record failed. The area was found non-accessible to a write request. |
| 0008 | | Error(s) detected on the SYMREC macro statement. A partial symptom record was recorded. Following are specific reasons why the symptom record component processed unsuccessfully: |

| Return Code | Reason Code | Explanation |
|---|---|---|
| | 0158 | Total length of the input symptom record exceeds the maximum. |
| | 015C | Optional segments of the input symptom record were found non-accessible. The record includes the accessible entries of the input symptom record. |
| 000C | | Serious error on the SYMREC macro statement. No symptom record was recorded. Following are specific reasons why the symptom record component processed unsuccessfully: |
| | 0104 | The first 2 bytes of the input symptom record does not contain the SR operand. |
| | 0108 | The input symptom record does not contain the required entries for section 2. |
| | 010C | The input symptom record does not contain the required entries for section 2.1. |
| | 0114 | The input symptom record does not contain the required entries for section 3. |
| | 0128 | Portions of the input symptom record were found non-accessible to a write request. |
| | 012C | Required portions of the input symptom record were found non-accessible to a write request. |
| | 0134 | Input symptom record address is in non-accessible storage. |
| | 0144 | Program attributes of the job issuing the SYMREC macro are not written in accordance with the symptom record component standards. |
| 0010 | | Serious error in the symptom record component. Error is not related to SYMREC macro statement. No symptom record was recorded. Following are specific reasons why the symptom record component processed unsuccessfully: |
| | 0F04 | Insufficient space in the LOGREC buffer to accommodate the symptom record. |
| | 0F08 | SYMREC macro service routine could not acquire storage for its workarea and a copy of the symptom record. |
| | 0F0C | Failure occurred while moving the symptom record to the LOGREC buffer. |
| | 0F10 | SYMREC macro service routine has a logic error. |
| 0014 | | Symptom record component is not operable. |

To invoke the SYMREC macro, an application must be authorized, enabled for interrupts, and running in primary addressing mode (bit 16 of the PSW is equal to zero). The AMODE attribute of the program can be set for either 24-bit or 31-bit addressing, but any address passed to SYMREC must be a 31-bit address. A program that invokes SYMREC can be running in cross memory mode, and it can be holding a suspend lock.

## The Format of the Symptom Record

The symptom record consists of six sections that are structured according to the format of the ADSR DSECT. These sections are numbered 1 through 5, including an additional section that is numbered 2.1. Because sections 2.1, 3, 4, and 5 of the symptom record are non-fixed, they do not need to be sequentially ordered within the record. In section 2, the application supplies the offset and the length of the non-fixed sections. The ADSR format is described in the *Debugging Handbook*, and the purpose of each section is as follows:

**Section 1 (Environmental data):** Section 1 contains the record header with basic environmental data. The SYMREC caller initializes this area to zero and stores the characters, 'SR', into the record header. The environmental data of section 1 is filled in automatically whenever the SYMREC macro is invoked. The environmental data that SYMREC stores in this section provides a system context within which the software errors can be viewed. Section 1 includes items such as:

- CPU model and serial numbers
- Date and time, with a time zone conversion factor
- Name of the customer installation
- Product ID of the control program
- Special features installed, if any

**Section 2 (Control data):** Section 2 contains control information with the lengths and offsets of the sections that follow. The application must initialize the control information before invoking SYMREC for the first time. Section 2 immediately follows section 1 in the symptom record structure.

**Section 2.1 (Component data):** Section 2.1 contains the name of the component in which the error occurred, as well as other specific component-related data. The application must also initialize section 2.1 before invoking SYMREC.

**Section 3 (Primary SDB symptoms):** Section 3 contains the primary string of problem symptoms, which may be used to perform tasks such as duplicate problem recognition. Whenever an application detects a given error, the string that it stores in section 3 becomes uniquely associated with that error incident as its **primary symptom string**. Note that an application does not store any primary symptom string or invoke SYMREC unless it detects an error in its own processing.

**Section 4 (Secondary SDB symptoms):** Section 4 contains an optional **secondary symptom string**. The purpose of the secondary string is to save any critical diagnostic values that existed at the time of the incident.

**Section 5 (Free-format data):** Section 5 contains logical segments of optional problem related data to aid in problem diagnosis. However, the data in section 5 is not in the SDB format, which is only found in sections 3 and 4. Each logical segment in section 5 is structured in a **key-length-data** format. This format consists of a two-byte key field, which is immediately followed by a two-byte length field, which is immediately followed by a variable length data field. The length that is indicated in the two-byte length field should be equal to the length of the data field.

## Symptom Strings — SDB Format

The symptom strings placed in sections 3 and 4 of the symptom record must be in the SDB (structured data base) format. In this format, the individual symptoms in sections 3 and 4 of the symptom record are separated by syntactical dividers called **prefixes**. For more information on the prefixes that SYMREC recognizes, see *Debugging Handbook*. Also see *A Structured Approach to Describing and Searching Problems* for more general information on symptom strings and prefixes. Examples of typical prefixes are:

| | |
|---|---|
| PIDS/ | a component name follows the slash |
| RIDS/ | a routine name follows the slash |
| AB/ | an abend code follows the slash |
| PRCS | a return code follows the slash |
| REGS/ | a register name follows the slash |
| LVLS/ | a release level follows the slash |
| FLDS/ | a data field name follows the slash |
| VALU/ | an error-related value follows the slash |

An SDB symptom string is a continuous character string. Each prefix must end in a slash, and a blank is required between successive symptoms in the string. For example:

```
LVLS/B10  PIDS/5752SASR  RIDS/ASRSERVX  PRCS/00000001  REGS/GR14
VALU/H81ABCDEF  FLDS/XSTATUS  VALU/CUSED
```

Symptom strings in SDB format can be interpreted as meaningful remarks about program incidents. The symptom string shown above is interpreted as follows:

| | |
|---|---|
| LVLS/B10 | In release level B10 |
| PIDS/5752SASR | of the component, 5752SASR, |
| RIDS/ASRSERVX | routine ASRSERVX was executing, |
| PRCS/00000001 | and it received return code 00000001. |
| REGS/GR14 | At that time, general register 14 |
| VALU/H81ABCDEF | contained hex 81ABCDEF, and |
| FLDS/XSTATUS | the field, XSTATUS, |
| VALU/CUSED | contained characters USED. |

## Using EREP and IPCS to Format Symptom Record Reports

To format and print various reports from the data on the repository, you use EREP and IPCS programs, which invoke the SYMREC post-processing services. The four basic reports generated by SYMREC post processing services are:

**The System Summary Report:** The system summary EREP report is an overview of processor, channel, subchannel, operating system, and I/O subsystem errors. The report contains a count per CPU of the symptom records that were stored.

**The Event History Report:** The event history report consists of one-line abstracts of selected information from each record, listed in chronological order. The one-line abstract is the primary symptom string. The event history also has a count of the symptom records classified by processor.

**The Detail Edit Report:** The detail edit report shows the entire contents of an error record, except for section 2. Optional fields that have not been completed, which contain hexadecimal zeroes, are not included in the report.

**The Detail Summary Report:** The detail summary report shows each unique primary symptom string in the repository; it does not duplicate symptom strings that recur. For each unique string, the report indicates the number of occurrences and the date and time of the first and last occurrence. SYMREC uses the first 80 bytes of the symptom string when comparing for duplicate strings.

**Note:** For detailed information on these reports, see *Debugging Handbook*.

Through EREP, you can generate all four reports, which are obtained when you specify the TYPE=S parameter. IPCS can generate only one kind of report, the detail edit report. To cause IPCS to generate this report, you specify the LOGDATA verb. LOGDATA causes one report to be generated for each symptom record in its recording buffer.

## Programming Notes for SYMREC Applications

This section contains programming notes on how the various fields of the ADSR data area (symptom record) are set. Some fields of the ADSR data area (symptom record) must be set by the caller of the SYMREC macro, and other fields are set by the system when the application invokes the SYMREC service. The fields that the SYMREC caller must always set are indicated by an RC code in the following sections. The fields that are set by SYMREC are indicated by an RS code.

The RA code designates certain flag fields that need to be set only when certain kinds of alterations and substitutions are made in the symptom record after the incident occurs. These alterations and substitutions must be obvious to the user who interprets the data. Setting these flag fields is the responsibility of the program that alters or substitutes the data. For example, if a program changes a symptom record that is already written on the repository, it should set the appropriate RA-designated flag-bit fields as an indication of how the record has been altered.

The remaining fields, those not marked by RC, RS, or RA, are optionally set by the caller of the SYMREC macro. When SYMREC is invoked, it checks that all the required input fields in the symptom record are set by the caller. If the required input fields are not set, SYMREC issues appropriate return and reason codes.

### Programming Notes for Section 1

Notes in this section pertain to the following fields, which are in section 1 of the ADSR data area.

```
ADSRID     Record header                          (RC)
ADSRGMT    Local Time Conversion Factor
ADSRTIME   Time stamp                             (RS)
ADSRTOD    Time stamp (HHMMSSTH)
ADSRDATE   Date (YYMMDD)
ADSRSID    Customer Assigned System/Node Name     (RS)
ADSRSYS    Product ID of Base System (BCP)        (RS)
ADSRCML    Feature and level of Symrec Service    (RS)
ADSRTRNC   Truncated flag                         (RS)
ADSRPMOD   Section 3 modified flag                (RA)
ADSRSGEN   Surrogate record flag                  (RA)
ADSRSMOD   Section 4 modified flag
ADSRNOTD   ADSRTOD & ADSRDATE not computed flag   (RS)
ADSRASYN   Asynchronous event flag                (RA)
ADSRDTP    Name of dump
```

**Notes:**

1. SYMREC stores the TOD clock value into ADSRTIME when the incident occurs. However, it does not compute ADSRTOD and ADSRDATE when the incident occurs, but afterwards, when it formats the output. When the incident occurs, SYMREC also sets ADSRNOTD to 1 as an indication that ADSRTOD and ADSRDATE have not been computed.
2. SYMREC stores the customer-assigned system node name into ADSRSID.
3. SYMREC stores the first four digits of the base control program component id into ADSRSYS. The digits are 5752, 5759 and 5745 respectively for MVS, VM, and DOS/VSE.
4. The ADSRDTP field is not currently used by the system.
5. If section 3 of a symptom record is changed or extended after SYMREC is invoked, the program that makes the change should set the ADSRPMOD flag to 1. If section 4 of a symptom record is changed or extended after SYMREC is invoked, the program that makes the change should set the ADSRSMOD flag to 1. The purpose of these flags is to indicate whether the symptom strings are original, intact, and exactly the same as when SYMREC was invoked.
6. If some application creates the record asynchronously, that application should set ADSRSYN to 1. 1 means that the data is derived from sources outside the normal execution environment, such as human analysis or some type of machine post-processing.
7. If SYMREC truncates the symptom record, it sets ADSRTRNC to 1. This can happen when the size of the symptom record provided by the invoking application exceeds SYMREC's limit.
8. ADSRSGEN indicates that the symptom record was not provided as 'first time data capture' by the invoking application. Another program created the symptom record. For instance, the system might have abended the program, and created a symptom record for it because the failing program never regained control. Setting the field to 1 means that another program surrogate created the record. The identification of the surrogate might be included with other optional information, for example, in section 5.
9. The application invoking SYMREC must provide the space for the entire symptom record, and initialize that space to hex zeroes. The application must also store the value 'SR' into ADSRID.
10. The fields ADSRCPM through ADSRFL2, which appear in the record that is written on the repository, are also written back into the input symptom record as part of the execution of SYMREC.

## Programming Notes for Section 2

Notes in this section pertain to the following fields, which are in section 2 of the ADSR data area.

| | | |
|---|---|---|
| ADSRARID | Architectural level designation | (RS) |
| ADSRL | Length of section 2 | (RC) |
| ADSRCSL | Length of section 2.1 | (RC) |
| ADSRCSO | Offset of section 2.1 | (RC) |
| ADSRDBL | Length of section 3 | (RC) |
| ADSRDBO | Offset of section 3 | (RC) |
| ADSRROSL | Length of section 4 | |
| ADSRROSA | Offset of section 4 | |
| ADSRRONL | Length of section 5 | |
| ADSRRONA | Offset of section 5 | |
| ADSRRISL | Length of section 6 | |
| ADSRRISA | Offset of section 6 | |
| ADSRSRES | Reserved for system use | |

**Notes:**

1. The invoking application must ensure that the actual lengths of supplied data agree with the lengths indicated in the ADSR data area. The application should not assume that the SYMREC service validates these lengths and offsets.
2. The lengths and offsets in section 2 are intended to make the indicated portions of the record indirectly addressable. Invoking applications should not program to a computed absolute offset, which may be observed from the byte assignments in the data area.
3. The value of the ADSRARID field is the architectural level at which the SYMREC service is operating. This field is supplied by the SYMREC service.
4. Section 2 has a fixed length of 48 bytes. Optional fields (not marked with RC, RS, or RA) will contain zeroes when the invoking application provides no values for them.

## Programming Notes for Section 2.1

Notes in this section pertain to the following fields, which are in section 2.1 of the ADSR data area.

| | | |
|---|---|---|
| ADSRC | C'SR21' Section 2.1 Identifier | (RC) |
| ADSRCRL | Architectural Level of Record | (RC) |
| ADSRCID | Component identifier | |
| ADSRFLC | Component Status Flags | |
| ADSRFLC1 | Non-IBM program flag | (RC) |
| ADSRLVL | Component Release Level | (RC) |
| ADSRPTF | Service Level | |
| ADSRPID | PID number | (RC) |
| ADSRPIDL | PID release level | (RC) |
| ADSRCDSC | Text description | |
| ADSRRET | Return Code | (RS) |
| ADSRREA | Reason Code | (RS) |
| ADSRPRID | Problem Identifier | |
| ADSRID | Subsystem identifier | |

**Notes:**

1. This section has a fixed length of 100 bytes, and cannot be truncated. Optional fields (not marked with RC, RS, or RA) will appear as zero if no values are provided.
2. ADSRCID is the component ID of the application that incurred the error, without the imbedded punctuation that normally appears when the component id is seen in print.

Under some circumstances, there can be more than one component ID involved. For ADSRCID, select the component ID that is most indicative of the source of the error. The default is the component ID of the detecting program. In no case should the component ID represent a program that only administratively handles the symptom record. Additional and clarifying data (such as, other component ID involved) is optional, and may be placed in optional entries such as ADSRCDSC of section 2.1, section 4, or section 5.

For example: if component A receives a program check; control is taken by component B, which is the program check handler. Component C provides a service to various programs by issuing SYMREC for them. In this case, the component ID of A should be given. Component B is an error handler that is unrelated to the source of the error. Component C is only an administrator. Note that, in this example, it was possible for B to know A was the program in control and the source of the program check. This precise definition is not always possible. B is the detector, and the true source of the symptom record. If the identity of A was unknown, then B would supply, as a default, its own component ID.

ADSRCID is not a required field in this section, although it is required in section 3 after the PIDS/ prefix of the symptom string. Repeating this value in section 2.1 is desirable but not required. Where the component ID is not given in section 2.1, this field must contain zeroes.

ADSRPID is the Program Information Department (PID) number assigned to the program that incurred the error. It appears as a seven-character value with no punctuation and one byte of padding. ADSRPID must be provided only by IBM programs that do not have an assigned component ID. Therefore, ADSRCID contains hex zeroes if ADSRPID is provided.

3. ADSRLVL is the release level of the assigned component ID. It is required even if the assigned component ID value is not given in ADSRCID for IBM products. All release level values are numeric values. Therefore, this field normally has a blank (X'40') as the rightmost pad character.
4. ADSRPIDL is the release level of the program designated by ADSRPID, and it should be formatted using the characters, V, R, and M as separators, where V, R, and M represent the version, release, and modification level respectively. For example, V1R21bbbbb is Version 1 Release 2.1 without any modification. No punctuation can appear in this field, and ADSRPIDL must be provided only when ADSRRPID is provided.
5. ADSRPTF is the service level. It may differ from ADSRLVL because the program may be at a higher level than the release. ADSRPTF can contain any number indicative of the service level. For example, a PTF, FMID, APAR number, or user modification number. This field is not required, but it should be provided if possible.
6. ADSRCDSC is a 32-byte area that contains text, and it is only provided at the discretion of the reporting component. It provides clarifying information. For example, 'IOS IOSB ANALYSIS ROUTINE'.
7. ADSRREA is the reason code, and ADSRRET is the return code from the execution of SYMREC. SYMREC places these values in registers 0 and 15 and in these two fields as part of its execution. The fields are right justified, and identical to the contents of registers 0 and 15.
8. ADSRCRL is the architectural level of the record. Note that ADSRARID (section 2) is the architectural level of the SYMREC service.
9. ADSRPRID is a value that can be used to associate the symptom record with other symptom records. This value must be in EBCDIC, but it is not otherwise restricted.

10. ADSRNIBM is a flag indicating that a non-IBM program originated the symptom record. When this flag is 1, ADSRPID, ADSRPIDL, and ADSRPTF are interpreted respectively as program name, program major level, and program fix level. For IBM programs originating a symptom record, this flag must be '0'.
11. ADSRSSID is the name of a subsystem. The primary purpose of this field is to allow IBM subsystems to intercept the symptom record from programs that run on the subsystem. They may then add their own identification in this field as well as additional data in sections 4 and 5. The subsystem can then pass the symptom record to the system via SYMREC. A zero value is interpreted as no name.
12. The ADSRVSE6 flag is set to 1 when the user has added another section to the symptom record after section 5 (that is, the contents of ADSRRISL are non-zero).

## Programming Notes for Section 3

Section 3 of the symptom record contains the primary symptoms associated with the error incident, and it is provided by the application that incurred the error, or some program that acts in its behalf. The internal format of the data in section 3 is the SDB format, with a blank separating each entry. Once this data has been passed to SYMREC by the invoker, it may not be added to or modified without setting ADSRPMOD to '1'. The data in this section is EBCDIC, and no hex zeros may appear. The symptoms are in the form K/D where K is a keyword of 1 to 8 characters and D is at least 1 character. D can only be an alphanumeric or @, $, and #.

**Notes:**

1. The symptom K/D can have no imbedded blanks, but the '#' can be used to substitute for desired blanks. Each symptom (K/D) must be separated by at least one blank. The first symptom may start at ADSRRSCS with no blank, but the final symptom must have at least one trailing blank. The total length of each symptom (K/D combination) can not exceed 15 characters.
2. This section is provided by the component that reports the failure to the system. Once a SYMREC macro is issued, the reported information will not be added to or modified, even if the information is wrong. It is the basis for automated searches, and even poorly chosen information will compare correctly in such processing because the component consistently produces the same symptoms regardless of what information was chosen.
3. If section 3 is modified, then a flag in section 1 (ADSRPMOD) should be set to warn users (such as detecting programs) that this data has been modified (perhaps to correct it), and may be useless in SDB automated searches.
4. The PIDS/ entry is required, with the component ID following the slash. It is required from all programs that originate a symptom record and have component a ID assigned. Further, it must be identical to the value in ADSRCID (section 2.1) if that is provided. (ADSRCID is not a required field).

## Programming Notes for Section 4

Section 4 of the symptom record contains the secondary symptoms associated with the error incident, and it is provided by the application that incurred the error, or some program that acts in its behalf. The internal format of the data in section 4 is the SDB format, with a single blank separating each entry. Once this data has been passed to SYMREC by the invoker, it may not be added to or modified without setting ADSRSMOD to 1. The data in this section is EBCDIC, and no hex zeroes may appear. The symptoms are in the form, K/D, where K is a keyword of one to eight characters and D is at least one character. D must be alphanumeric or @, $, or #.

**Notes:**

1. The secondary symptom string is in the same SDB format as the primary symptom string.
2. If any changes are made in this section, ADSRSMOD must be set to 1.

## Programming Notes for Section 5

Section 5 of the symptom record contains logical segments of data that are provided by the component or some program that acts in its behalf. The component may store data in section 5 when SYMREC is invoked, or the system may add notes in this section at the time of SYMREC execution. Further, section 5 may be added to by a general edit of the record or by other programs operating on the entry any time after SYMREC is invoked.

**Notes:**

1. The first segment must be stored at symbolic location ADSR5ST. In each segment, the first two characters are a hex key value, and the second two characters are the length of the data string, which must immediately follow the two-byte length field. Adjacent segments must be packed together. The length of section 5 is in the ADSRRONL field in section 2, and this field should be correctly updated as a result of all additions or deletions to section 5.
2. There are 64K key values grouped in thirteen ranges representing thirteen potential SYMREC user categories. The data type (that is, hexadecimal, EBCDIC, etc.) of section 5 is indicated by the category of the key value. Thus, the key value indicates both the user category and the data type that are associated with the information in section 5. Because the component ID is a higher order qualifier of the key, it is only necessary to control the assignment of keys within each component ID or, if a component ID not assigned, within each PID number.

| Key Value | User Category and Data Type |
|---|---|
| 0001-00FF | Reserved |
| 0100-0FFF | MVS system programs |
| 1000-18FF | VM system programs |
| 1900-1FFF | DOS/VSE system programs |
| 2000-BFFF | Reserved |
| C000-CFFF | Product programs and non-printable hex data |
| D000-DFFF | Product programs and printable EBCDIC data |
| E000-EFFF | Reserved |
| F000 | Any program and printable EBCDIC data |
| F001-F0FF | Not assignable |
| F100-FEFF | Reserved |
| FF00 | Any program and non-printable hex data |
| FF01-FFFF | Not assignable |

## Obtaining Accumulated Processor Time

The TIMEUSED macro offers you the opportunity to record execution times and to measure performance. TIMEUSED returns the amount of processor time that a work unit (such as a task or an SRB) has used since it began executing.

TIMEUSED is available only to authorized programs (supervisor state or PSW key 0-7).

*Example of measuring performance with TIMEUSED macro:*

Use TIMEUSED to measure the efficiency of a routine or other piece of code. If you need to sort data, you may now code several different sorting algorithms, and then test each one. The logic for a test of one algorithm might look like this:

1. Issue TIMEUSED
2. Save old time
3. Run sort algorithm
4. Issue TIMEUSED
5. Save new time
6. Calculate time used (new time - old time)
7. Issue a WTO with the time used and the algorithm used.

After running this test scenario for all of the algorithms available, you can determine which algorithm has the best performance.

# Communication

The following types of communication are included in this chapter:

- Interprocessor communication
- Writing operator messages
- Inter-address space communication

## 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.*

### Service Classes

IPC services divide the SIGP order codes into two classes, direct and remote. The SIGP instruction and the valid order codes are documented in *Principles of Operation.*

Direct service class is defined for those control program functions that require the modification or sensing of the physical state of one of the configured processors. The following SIGP order codes can be invoked via the DSGNL macro instruction.

**Sense:** The specified processor presents its status to the issuing processor. No other action is caused at the specified processor.

**Start:** The specified processor is placed in the operating state. The processor does not necessarily enter the operating state during the execution of the SIGP instruction. No action is caused at the specified processor if that processor is in the operating state when the order code is accepted.

**Stop:** The specified processor stops. The processor does not necessarily enter the stopped state during the execution of the SIGP instruction. No action is caused at the specified processor if that processor is in the stopped state when the order code is accepted.

**Restart:** The specified processor restarts. The processor does not necessarily perform the function during the execution of the SIGP instruction.

**Stop and Store Status:** The specified processor stops and stores status. The processor does not necessarily complete the operation, or even enter the stopped state, during the execution of the SIGP instruction.

**Store Status at Address:** The specified processor stores status starting at a specified location. If the specified processor is not stopped, it does not accept the order. The processor does not necessarily complete the operation during the execution of the SIGP instruction.

**Initial CPU Reset:** The specified processor performs initial processor reset. The execution of the reset does not affect other processors and does not cause any channels, including those reconfigured to the specified processor, to be reset. The reset operation is not necessarily completed during the execution of the SIGP instruction.

**CPU Reset:** The specified processor performs processor reset. The execution of the reset does not affect other processors and does not cause any channels, including those configured to the specified processor, to be reset. The reset operation is not necessarily completed during the execution of the SIGP instruction.

**Set Prefix:** The specified processor's prefix register is set to the value passed to it by the control program. If the specified processor is not stopped, it does not accept the order. This function is not necessarily completed during the execution of the SIGP instruction.

Remote class services are defined for those control program functions that require the execution of a software function on one of the configured processors. The remaining SIGP functions are defined as remote services. External call is a remote pendable service that can be invoked via the RPSGNL macro and emergency signal is a remote immediate service that can be invoked via the RISGNL macro. A description of these services follows:

**External Call:** An "external call" external-interruption condition is generated to 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. Only one external-call condition can be kept pending in a processor at a time. Issue the RPSGNL macro instruction to invoke the external-call function.

**Emergency Signal:** 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 instruction to invoke the emergency signal function.

## Status Indicators

If the user receives a return code of 8 from the DSGNL macro, register 0 contains a status indicator describing the state of the specified processor. The status indicators describe the following conditions:

**Equipment Check:** This condition exists when the processor executing an instruction detects equipment malfunctioning that has affected only the execution of the instruction and the associated hardware function. The order code may or may not have been transmitted, and may or may not have been accepted, and the status bits provided by the specified processor may be in error.

**Incorrect State:** This condition exists when an order has been directed to a processor that is not stopped. The condition, when present, is indicated only in response to status or prefix.

**Invalid Parameter:** This condition exists when an address exception occurs. This happens when the storage referenced by the status or prefix function is not installed or not configured on the system. The condition, when present, is indicated only in response to status or prefix.

**External Call Pending:** This condition exists when an external-call interruption condition is pending in the specified processor because of a previously issued SIGP instruction. The condition exists from the time an external-call function is accepted until the resulting external interruption is accepted. The condition may exist on the issuing processor or another processor. The condition, when present, is indicated only in response to sense and to external call.

**Stopped:** This condition exists when the specified processor is in the stopped state. The condition, when present, is indicated only in response to sense.

**Operator Intervening:** This condition exists when the specified processor is executing certain operations initiated from the console or the remote operator control panel. The particular manually initiated operations that cause this condition to be present depend on the model and on the specified functions. This condition, when present, can be indicated in response to all functions. Operator intervening is indicated in response to sense if the condition is present and precludes the acceptance of any of the installed orders (or SIGP hardware functions). The condition might also be indicated in response to unassigned or uninstalled orders.

**Check Stop:** This condition exists when the specified processor is in the check-stop state. The condition, when present, is indicated only in response to sense, external call, emergency signal, start, stop, restart, and stop and store status. The condition may also be indicated in response to unassigned or uninstalled functions.

**Invalid Function:** This condition exists during the communications associated with the execution of SIGP when the specified processor decodes an unassigned or uninstalled function code.

**MSSF Failure:** This condition exists when the MSSF (3082) is currently inoperative. The MSSF performs the SIGP between two processors.

**Receiver Check:** This condition exists when the specified processor detects malfunctioning of equipment during the communications associated with the execution of SIGP. When this condition is indicated, the function has not been initiated and, because the malfunction may have affected the generation of the remaining receiver status bits, these bits are not necessarily valid. A machine-check condition may or may not have been generated at the specified processor.

# Writing and Deleting Messages (WTO, WTOR, DOM, and WTL)

The WTO and WTOR macro instructions allow you to write a message to a display device or a printer at the operator console. Besides writing a message, WTOR allows you to request a reply from the operator who receives the message. The DOM macro instruction allows you to delete a message that is already written to the operator. The standard printable EBCDIC characters that constitute messages are shown in Figure 21. All other characters, which are not printable, are replaced by blanks. If the terminal does not have dual-case capability, it prints lowercase EBCDIC characters as uppercase EBCDIC characters.

| Hex Code | EBCDIC | Hex Code | EBCDIC | Hex Code | EBCDIC | Hex Code | EBCDIC |
|---|---|---|---|---|---|---|---|
| 40 | (space) | 7B | # | 99 | r | D5 | N |
| 4A | ¢ | 7C | @ | A2 | s | D6 | O |
| 4B | . | 7D | ' | A3 | t | D7 | P |
| 4C | < | 7E | = | A4 | u | D8 | Q |
| 4D | ( | 7F | " | A5 | v | D9 | R |
| 4E | + | 81 | a | A6 | w | E2 | S |
| 4F | \| | 82 | b | A7 | x | E3 | T |
| 50 | & | 83 | c | A8 | y | E4 | U |
| 5A | ! | 84 | d | A9 | z | E5 | V |
| 5B | $ | 85 | e | C1 | A | E6 | W |
| 5C | * | 86 | f | C2 | B | E7 | X |
| 5D | ) | 87 | g | C3 | C | E8 | Y |
| 5E | ; | 88 | h | C4 | D | E9 | Z |
| 5F | ¬ | 89 | i | C5 | E | F0 | 0 |
| 60 | - | 91 | j | C6 | F | F1 | 1 |
| 61 | / | 92 | k | C7 | G | F2 | 2 |
| 6B | , | 93 | l | C8 | H | F3 | 3 |
| 6C | % | 94 | m | C9 | I | F4 | 4 |
| 6D | _ | 95 | n | D1 | J | F5 | 5 |
| 6E | > | 96 | o | D2 | K | F6 | 6 |
| 6F | ? | 97 | p | D3 | L | F7 | 7 |
| 7A | : | 98 | q | D4 | M | F8 | 8 |
| | | | | | | F9 | 9 |

Figure 21. EBCDIC Characters Printed or Displayed on an MCS Console

**Notes:**

1. If the display service or printer is defined to JES3, the following characters are translated to blanks:

   ┬ ! ; ¬ : "

2. The system recognizes the following hexadecimal representations of the U.S. national characters: @ as X'7C'; $ as X'5B'; and # as X'7B'. In countries other than the U.S., the U.S. national characters represented on terminal keyboards might generate a different hexadecimal representation and cause an error. For example, in some countries the $ character generates a X'4A'.

## Routing the Message

The ROUTCDE parameter allows you to specify the routing code or codes for a WTO and WTOR message. The routing codes determine which MCS console or consoles receive the message. Each code represents a predetermined subset of the consoles that are attached to the system, and that are capable of displaying the message. It is up to the user to define the consoles that belong to each routing code. WTO and WTOR allow routing codes from 1 to 128. Routing codes 29 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.

The parameters, MSGTYP and MCSFLAG, which are associated with message routing, should only be used by programmers familiar with multiple console support (MCS). The MSGTYP parameter is typically used for messages related to the monitor command. The MCSFLAG parameter is used to specify various attributes of the message, such as:

- Whether the message is for a particular console
- Whether the message is for all active consoles
- Whether the message is a command response
- Whether the message is for the hardcopy log

The MCSFLAG=BUSYEXIT parameter determines what happens if no message buffers are available. If BUSYEXIT is specified and no console buffers for either MCS or JES3 are available, or if BUSYEXIT is specified and there is a JES3 WTO staging area excess, the WTO is terminated. If BUSYEXIT is not specified, the WTO invocation will be placed in a wait state until WTO buffers are again available. BUSYEXIT is available to authorized programs only.

When MSGTYP=Y is specified, the MCSFLAG field indicates that the MSGTYP field is to be used for the message routing criteria. In this case, the issuer of the WTO(R) must set a message identifier bit in the MSGTYP field of the macro expansion. The macro expansion is mapped by IEZWPL. When a message identifier bit is recognized by the system, the message is routed to all consoles and TSO terminals (in operator modes) that have requested the particular type of information represented by the identifier bit. If there are no consoles or terminals requesting that kind of information, a WTO message is not sent anywhere; however, a WTOR message is sent to the master console. The routing codes and REG0 MCSFLAG field, if present, are ignored when MSGTYP=Y is specified.

Another alternative for routing a message is to use the CONSID parameter. This parameter lets you specify a field or register that contains the four-byte id of the console that is to receive the message. This is a handy alternative to the MCSFLAG option of placing the console id in register zero. When you issue a WTO or WTOR macro that uses both the CONSID and the ROUTCDE parameters, the message(s) will go to all of the consoles specified by both parameters.

**Notes:**

1. By using the various parameters of WTO(R), messages can be routed by route code, console id, descriptor code, and message type. Messages can be sent on multiple paths. See the description of the WTO(R) macro instruction in Volume 2 for additional information.
2. 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 D R console command, which operators can issue at the console. Use the KEY parameter on the WTO or WTOR macro for this purpose.

## Writing a Multiple-Line Message

Messages consisting of multiple lines should be issued using the WTO multiple-line capability to assure that all lines of a multiple-line message appear together and are not broken up by other single-line messages.

Using the WTO macro instruction, a program can write a multiple-line message to one or more operator consoles. System programs (supervisor state, PSW key 0-7, or APF-authorized) can create a message that consists of up to 255 lines with one WTO request. If more than 255 lines are needed, the authorized user can issue more than one WTO.

When issuing more than one request, the first WTO supplies the first lines of the message, up to a limit of 255 lines. Subsequent WTO requests can then add lines to the message. The additional lines appear at the end of the message and continue until an "END" line is specified. For the first request, you must ensure that the left most three bytes of register zero are zero. If the bytes are not 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 via the CONNECT parameter of WTO.

## Embedding Label Lines in a Multiline 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 multiline WTO message for a tabular display.

System programs (supervisor state, PSW key 0-7, or APF-authorized) that are authorized to add lines to an existing multiline WTO message are also permitted to embed label lines within that existing multiline 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.

**Note:** You cannot use the WTO macro instruction to embed label lines. The WTO macro instruction handles label lines at the beginning of the message only.

## Using the Authorized Parameters of WTO and WTOR

CONNECT, JOBID, JOBNAME, SUBSMOD, SYSNAME, PRTY, and SYSNAME are authorized parameters used with WTO or WTOR.

The CONNECT parameter is used to connect a subsequent message to a previous message. For example, if your program develops a large, multi-line message of unknown length, it can issue different WTOs for the different parts of the message at different times. The CONNECT parameter can force all these WTOs to use the same message id, and cause all the different parts of the message to be physically reunited at the display console as a single message. To use CONNECT, you save the message id that is returned in general register 1 after you issue the first part of the multi-line message. On subsequent invocations of WTO for the remaining message parts, you supply the returned message id as an input parameter by using the CONNECT parameter. The end-of-message character in the text allows the system to recognize the last WTO in the sequence, and to start issuing the message. CONNECT is mutually exclusive with CONSID and SYSNAME, and it is not available with WTOR.

The SYSNAME parameter, which is mutually exclusive with CONNECT, is used to provide an eight-byte system name that appears on the console with the message. SYSNAME is available with WTO and WTOR.

The WQEBLK parameter, which is similar in principle to the DOMCBLK parameter of the DOM macro, is used when the control information for WTO(R) is in a table instead of in the input parameters. WQEBLK is mutually exclusive with all other parameters. As an example of how WQEBLK can be used, an application might capture the internally-generated control table resulting from the invocation of a previous WTO(R). Then it might supply this table as an input parameter in the subsequent WTO(R), by using the WQEBLK parameter.

The JOBNAME and JOBID parameters are used to correlate WTO or WTOR macros and their resulting messages with the jobs that are passing through the system.

The SUBSMOD parameter is used to indicate whether the message can be modified by a subsystem, and the PRTY parameter is used to give the message a priority that is visible when the message appears on the console. These two parameters are only available with WTOR.

## 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. The actual message, however, remains displayed until it rolls off the screen. When DOM deletes a message, it does not actually erase the message. It only changes its format, displaying it like a non-action message.
- If the message is not yet on the screen, DOM deletes the message before it appears. The DOM processing does not affect the logging action. That is, if the message is supposed to be logged, it will be, regardless of when or if a DOM is issued. The message is logged in the format of a message that is waiting for operator action.

The program that generates an action message is responsible for deleting that message.

## Identifying Messages to be Deleted

To identify the message(s) that you want to delete, you normally use the MSG, MSGLIST, or TOKEN parameters. When you issue a WTO or WTOR macro instruction to write a given message to the operator, the system generates a message id, which it returns in general register 1. To delete the message, you can issue the DOM macro instruction with a MSG or MSGLIST parameter specifying the same system-generated message id that WTO or WTOR returned in general register 1. If you specify MSGLIST (message list), then several message ids can be associated with the delete request. The number of message ids in the message list is defined by the COUNT parameter or it is defined by an 1 in the high order bit position of the last message id in the list. The count parameter cannot exceed 60.

On the other hand, the TOKEN parameter allows the message id to be generated by the user rather than the system. When you issue WTO or WTOR with a TOKEN parameter, the system associates your TOKEN parameter with all the message(s) that are written by this particular WTO or WTOR. Then you can issue DOM with the same TOKEN parameter to delete all the message(s) associated with the token.

## Limiting the Extent of Message Deletion

DOM allows you to limit the extent of message deletion. One way to limit the extent of message deletion is to use the SYSID parameter, which deletes only messages from a particular system whose id you specify. Another way to limit the extent of deletion is to use the SCOPE parameter. If you specify SCOPE=SYSTEMS, the delete request is sent to all other processors. To delete messages only within the host system, you can specify SCOPE=SYSTEM. SYSID and SCOPE can only be coded by authorized users.

### Custom-Built Delete Functions

When DOM executes, it builds a control block from the specified parameters to control the execution of the delete. All the input parameters specified on the DOM macro are represented by fields in this control block. However, the control block can be built directly by an authorized user. To build the DOM control block without specifying parameters, simply specify the address of the control block by using the DOMCBLK parameter. When this parameter is specified, the system does not build any delete control block, and it substitutes the user-provided control block instead.

**Note:** Specifying the REPLY= parameter of the DOM macro causes an MNOTE warning message to be issued at assembly time. The MNOTE warns you that you are coding the REPLY= parameter, which is a function no longer supported in the system. If you code the REPLY= parameter and receive the MNOTE warning, remove the REPLY= parameter from your program and reassemble it. Programs containing the REPLY= parameter that are already assembled do not need to be reassembled.

## Writing to the System Log

There are two ways to request that the system write a message to the system log:

- Use the HRDCPY option on the MCSFLAG parameter on the WTO macro.
- Use the HARDCOPY statement in CONSOLxx member of SYS1.PARMLIB to specify that the message appear at the device that is the hardcopy log. Note that you can use this member to direct the system log to a printer or a spooled file.

  You can change this specification through the VARY HARDCPY command.

IBM recommends that you do not use the WTL macro to write to the system log. This macro generates messages with formats that are inconsistent with other messages in the log.

# Inter-Address Space Communication

There are many advantages to the use of multiple virtual address spaces. Virtual addressing permits an addressing range that is greater than the real 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 integrity 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 addition, having a separate address space for data increases the amount of data that can be addressed.

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 chapter (see "Asynchronous Address Space Communication.")
- Using cross memory services, a synchronous process that is also described later in this chapter (see "Cross Memory" on page 1-79.)

## Asynchronous Address Space Communication

A program can use a dispatchable unit of work, the SRB, for asynchronous communication between the program and a routine, the **SRB routine**, in another address space (or a routine in the same address space). This process is called **scheduling an SRB**. While WAIT and POST macros can synchronize communication between the program and the SRB routine, the major advantage of scheduling an SRB is that an SRB routine is asynchronous in nature and executes independently of the routine that scheduled it. 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 executing:

- To process in parallel

  In a multi-processor environment, the SRB routine, after being scheduled, can be dispatched on another processor and can execute concurrently with the routine that scheduled it. The scheduling program can continue to do other processing in parallel with the SRB routine.

- 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 executing.

- To make changes of state

  In some instances, a routine might be executing in some state that prevents certain functions from being performed. (For example, a routine that is in a disabled state cannot request a suspend-type lock.) A routine can avoid these restrictions by scheduling an SRB to complete the function.

- 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 execute at a priority higher than that of any address space or at the priority of the address space in which it is scheduled.

### A Service Request Block (SRB)

An SRB is a control block that represents an SRB routine that performs a particular function or service in a specified address space. The SRB is similar to a TCB in that it identifies a unit of work to the dispatcher. Some characteristics of an SRB are:

- The SRB is built by the program.
- The SRB is required only for initial dispatch. The user can free or reuse the SRB after it is dispatched.
- 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 an FRR stack.

Two macros schedule and manage SRBs:

- The **SCHEDULE macro** service places the SRB on a dispatcher queue to be dispatched when it becomes the highest priority work in the system. When the system dispatches the SRB, the SRB routine begins executing.
- The **PURGEDQ macro** service allows for cleanup of SRB activity.

The scheduling program must be in supervisor state with PSW key 0. It first allocates storage for the SRB from commonly addressable fixed storage (for example, subpool 245) either above or below 16 megabytes. The storage key should be 0. The program then initializes fields in the SRB that identify:

- The SRB routine
- The address space in which the SRB routine is to execute
- The priority level of the SRB relative to other requests in the system
- Additional information for recovery and control

The SRB can be reused after it has been dispatched. The program must provide the serialization to ensure that it doesn't reschedule an SRB, or change or free the SRB while it still on a dispatcher queue.

It is the scheduling program's responsibility (not the system's) to obtain storage for the SRB, and then to free this storage when the SRB is no longer needed.

## The Content of an SRB

Before issuing the SCHEDULE macro, the scheduling program must initialize the fields in the SRB. Use the following information to help you initialize the SRB. IHASRB macro maps the structure of an SRB. To see the format of the SRB, see IHASRB mapping macro in *Debugging Handbook*. You can include IHASRB in your program.

SRBASCB — Contains the address of the ASCB of the address space in which the SRB routine will execute.

SRBPKF — Indicates, in the 4 high-order bits, the PSW key of the SRB routine. The 4 low-order bits must be zero.

SRBEP — Specifies the address of the entry point of the SRB routine. If the SRB routine is to execute in 31-bit addressing mode, set the high-order bit in the field to 1; if the routine is to execute in 24-bit addressing mode, set the high-order bit to 0.

SRBSAVE — Contains all zeroes. This field is used by the system.

SRBPARM — Contains the address of a user parameter area. The system will load the address into register 1 when the system dispatches the SRB routine. Through this field, the scheduling program passes information to the SRB routine.

SRBCPAFF — Defines the processor affinity. If all zeroes or all ones, no affinity is implied. Otherwise, this field contains a bit mask in which the bits that are set "on" indicate on which processors the SRB can be dispatched. (For example, set the nth bit "on" to indicate that the SRB can be dispatched on the processor with physical address n.)

SRBRMTR — Contains the address of an RMTR. This routine is responsible for cleaning up an SRB that has been scheduled but not yet dispatched. The RMTR is required; SRBRMTR must contain a valid nonzero address. For information about the RMTR, see "Resource Manager Termination Routine (RMTR)" on page 1-78.

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 abends and its FRR does not exist or does not retry, the task is scheduled for abnormal termination.
- If the specified TCB 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.

SRBFRRA — Contains the address of an FRR that receives control if the SRB routine abends. If the FRR is to execute in 31-bit addressing mode, set the high-order bit in the field to 1; if the routine is to execute in 24-bit addressing mode, set the high-order bit to 0.

## 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. 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 it is dispatched, 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, it is recommended that you specify SCOPE=LOCAL (the default).

## Characteristics and Restrictions of SRB Routines

At entry, an SRB routine is in supervisor state, primary ASC mode, enabled and unlocked. The general purpose registers contain the following:

| Register | Contents |
|---|---|
| 0 | Address of the SRB |
| 1 | Area for passing user parameters from the scheduling program to the SRB routine (same as SRBPARM) |
| 2 | If FRR=YES, 24-bit address of FRR parameter area; otherwise unpredictable |
| 14 | Return address |
| 15 | Entry point address |

Other general purpose registers and all access registers are unpredictable.

The SRB routine runs in the operating mode known as **SRB mode**. Code in SRB mode:

- Cannot leave supervisor state and must establish its own recovery environment. However, the scheduling program can specify that the SRB routine be dispatched with a LOCAL lock held (LLOCK=YES) or have a recovery routine established for the SRB routine (FRR=YES), 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. The macro descriptions in the *SPL: System Macros and Facilities Volume 2* and *Supervisor Services and Macro Instructions* 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 callers.
- Must provide for all cleanup before it completes execution. Cleanup activity might include freeing the SRB storage.
- Must return control to the address supplied in register 14, in supervisor state with no locks held, except the CPU lock. (If LLOCK = YES, the routine must release the LOCAL lock.)
- Can issue a PC instruction and schedule an SRB
- Should not be a long-running program. An SRB routine is generally not preempted by I/O interruptions once the SRB is dispatched.

Although SRB routines run enabled and can be interrupted by an asynchronous interruption, they do not lose control to higher priority tasks or SRBs until they give up control voluntarily. However, SRBs 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 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, which allows the system to dispatch other work on the active processor. The system redispatches the SRB when after 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 continues to hold those locks. The system suspends other workunits that require the lock held by the suspended SRB until the system redispatches the SRB and explicitly releases those locks.

## Purging SRBs (PURGEDQ)

Because an SRB routine is dispatched after the program actually issues the SCHEDULE macro, the conditions that existed in the system at the time the SCHEDULE was issued might have changed by the time the SRB routine is dispatched. 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. For task termination, any SRBs associated with the task (SRBPTCB) are purged. For address space termination, any SRBs scheduled to the address space (SRBASCB) are purged. If there are any other conditions for which your SRBs should be purged, you should issue PURGEDQ to cover them. For this reason, a program, such as an FRR, 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

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 of 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.

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 current address space, the PURGEDQ routine waits for completion of any active SRBs and then 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. No locks should be held when PURGEDQ is invoked.
- If you specify an address space other than the current address space, only SRBs that have not yet been dispatched are affected because PURGEDQ does not wait for SRBs already dispatched but not completed.

If you omit the ASID parameter, the system uses the current address space.

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 current address space.

Specify the ASIDTCB parameter in one of the following ways:

1. To purge all SRBs scheduled to a specific address space as defined by ASID:

| | | |
|---|---|---|
| Bytes 0 - 7 | Zero | The system is to purge all SRBs defined by the ASID (SRBASCB) and RMTR parameters, regardless of their task (SRBPASID) and address space (SRBPTCB) association. |

2. To purge all SRBs scheduled by a specified address space:

| | | |
|---|---|---|
| Bytes 0 - 1 | Reserved | The system is to purge all SRBs defined by the ASID and RMTR parameters and associated with the specified address space (SRBPASID), regardless of their task (SRBPTCB). |
| Bytes 2 - 3 | ASID | |
| Bytes 4 - 7 | Zero | |

3. To purge SRBs associated with a specified TCB in a specified address space:

| | | |
|---|---|---|
| Bytes 0 - 1 | Zero | The system is to purge all SRBs defined by the ASID and RMTR parameters and associated with the specified address space (SRBPASID) and task (SRBPTCB). (If you specify SRBPTCB, you must also specify SRBPASID.) |
| Bytes 2 - 3 | ASID | |
| Bytes 4 - 7 | TCB | |

All other values produce unpredictable results.

### 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 activity. The routine can either free the SRB storage by invoking the FREEMAIN 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 must be enabled, in supervisor state, with PSW key 0, and hold no locks. Entry register contents are as follows:

| Register | Contents |
|---|---|
| 0 | Contents of register 0 of the caller of PURGEDQ at the time the PURGEDQ SVC was issued. This register allows the caller of PURGEDQ to pass information to the RMTR. |
| 1 | Address of the dequeued SRB. |
| 2 | Contents of SRBPARM of the dequeued SRB. |
| 14 | Return address of PURGEDQ. |
| 15 | Entry point of RMTR. |

The RMTR must return control using a BR 14, enabled, in supervisor state with PSW key 0 and hold no locks. It may, however, acquire locks, issue SVCs and destroy input registers during its processing.

## Synchronous Inter-Address Space Communication

MVS/XA provides a synchronous method of communication between address spaces that is called cross memory. Using cross memory, programs can pass control directly to programs in other address spaces and can move data directly from one address space to another. At any time, a program has associated with it two addressable address spaces, the primary address space and the secondary address space. It is between these two address spaces, which may be the same, that synchronous communication occurs.

Three of the ways cross memory can be used are program sharing, data movement, and data access.

**Program Sharing:** A program residing in the private area of a particular address space can be directly called by programs residing in a number of different address spaces. These address spaces must have such calling ability defined for them. MVS/XA provides a set of macro instructions to establish the required access structures. Thus, a service needed by a number of address spaces no longer has to reside in commonly addressable storage, or have multiple images in many address spaces.

**Data Movement:** A program can move data directly between the primary and secondary address spaces, within an address space, and between storage areas of differing storage protection keys. Thus, programs can pass data directly between address spaces.

**Data Access:** A program residing in commonly addressable storage can choose to access data from either the primary or secondary address space. Being able to access data in two address spaces increases the amount of data that the program can handle.

Program sharing, data movement, and data access enable cross memory to provide:

- Storage isolation and protection of code and data structures. By moving their programs and data structures from commonly addressable storage to their own address spaces, MVS/XA components and subsystems avoid the accidental destruction of their information by unrelated processes.
- Migration of code and data from commonly addressable storage to private storage. With code and data migrating to private storage, the common storage requirements of the system decrease. This decrease effectively expands the private area addressing range and provides each user with more virtual storage.
- Creation of data address spaces. Partitioning data into address spaces isolates sensitive data and provides restricted access to it. This also enables a program to address greater amounts of data.

## Cross Memory

Cross memory is a very complex concept, and there are a number of warnings and restrictions associated with its use. Before listing the restrictions, however, some definitions are needed.

**Cross Memory Terminology:** The following terms are associated with cross memory.

- Cross memory environment: The environment in which synchronous inter-address space communication can take place.
- Home address space: The home address space, whose address space identifier (ASID) is called the HASID[1], is the address space defined by PSAAOLD. The home address space contains the address space local control blocks that describe a unit of work to the control program. On initial dispatch of a unit of work, the home address space and the primary address space are the same.
- Primary address space: The primary address space, whose ASID is called the PASID[2], is the address space whose segment table is used to access data and instructions in primary mode.
- Secondary address space: The secondary address space, whose ASID is called the SASID[2], is the address space whose segment table is used to access data in secondary mode. In secondary mode, instructions must be in common storage because they might be fetched from either the primary or secondary address space.
- Current address space: The current address space is the primary address space when in primary mode and the secondary address space when in secondary mode.
- Primary mode: In primary mode, instructions and data are fetched from the primary address space.
- Secondary mode: In secondary mode, data is fetched from the secondary address space. Instructions might be fetched from either the primary or secondary address space.
- Home mode: A unit of work is in home mode if it is in primary mode and HASID and PASID are the same.
- CML lock: The cross memory local (CML) lock is the local level lock of an address space other than the home address space.

---

1 *Principles of Operation* uses the terms HASN, PASN, SASN, and primary-space mode. MVS/XA publications use the equivalent terms HASID, PASID, SASID, and primary mode.

2 *Principles of Operation* uses the terms HASN, PASN, SASN, and primary-space mode. MVS/XA publications use the equivalent terms HASID, PASID, SASID, and primary mode.

- Cross memory mode: Cross memory mode exists when at least one of the following is true:

  The home address space is not the primary address space.
  The home address space is not the secondary address space.
  Secondary mode is active.
  A CML lock is held.

- Active addressing bind to an address space: An executing unit of work has an active addressing bind to an address space if that address space is the current PASID or SASID.
- Active bind to an address space: An executing unit of work has an active bind to an address space if the unit of work holds the CML lock of that address space or if the address space is associated with the current HASID, PASID, or SASID.

A series of macro instructions create a cross memory environment. The macro instructions establish the necessary linkage and authorization information for synchronous inter-address space communication. The following System/370-XA instructions actually accomplish the communication:

- PC - program call - causes another program to get control. The program can be in another address space.
- PT - program transfer - returns control from the program called by the PC instruction to the calling program.
- SSAR - set secondary ASN[3] - sets the secondary address space to any desired address space.
- MVCP - move to primary - moves data from the secondary address space to the primary address space.
- MVCS - move to secondary - moves data from the primary address space to the secondary address space.
- MVCK - move with key - moves data between storage areas that have different protection keys.
- SAC - set address space control - explicitly sets either the primary or secondary mode.
- IAC - insert address space control - indicates in a general purpose register whether primary or secondary mode is in effect.
- EPAR - extract primary ASN[3] - places the primary ASID into a general purpose register.
- ESAR - extract secondary ASN[3] - - places the secondary ASID into a general purpose register.

**Warnings and Restrictions:** The design and implementation of programs using synchronous cross memory communication is extremely complex. System services use cross memory on a user's behalf; the user can obtain the benefits of cross memory without having to know the details. Using cross memory services improperly could cause severe system problems. Therefore, it is very important to consider all the implications of using cross memory. Some general considerations that apply to users of cross memory are:

- Real storage requirements might increase.
- Resource management is different.
- Accounting methods might be affected.

---

[3] *Principles of Operation* uses the term ASN. MVS/XA publications use the equivalent term ASID.

Cross memory has the following specific restrictions:

- Services are not available in cross memory mode unless their description specifically states that they are available.
- Code running in cross memory mode cannot issue any SVCs except ABEND. That is, any system service that depends on SVCs is not available in cross memory mode. TSO test, for example, provides only limited testing for programs that execute in cross memory mode because TSO test uses the TEST SVC.
- Only one step of a job can establish ownership of space switch entry tables. Subsequent job steps cannot issue the LXRES, AXRES, or ETCRE macro instructions.
- MVS/XA does not support cross memory accesses to a swapped-out address space; such accesses cause an ASID translation exception-program interruption that is treated as an error. Thus, in order to be accessed, the address space must be one of the following:

  The home address space
  A non-swappable address space
  An address space whose local lock is held

  This restriction must be a major consideration when using cross memory because it might increase the storage requirements of the system.
- Some MVS/XA services require an active addressing bind to the address space in which processing is to occur. Such an address space must be one of the following:

  The home address space
  A non-swappable address space
  An address space whose local lock is held

  If none of these three requirements are met, the address space might be swapped out and a unit of work that referenced the address space would be abnormally terminated.
- Storage acquired in a cross memory environment is attributed to the job step task of the address space in which it was obtained if the subpool it comes from is task related. A program that acquires such a resource should provide a task termination/address space termination resource manager to clean up any resources obtained on behalf of the terminating task or address space but attributed to another address space's job step task. For more considerations on resource management see "Designing a PC Routine" later in this section.
- Execution time is attributed to the home address space, not necessarily the address space in which the program executes.
- Routines that get control as the result of a PC instruction must not use the checkpoint/restart facility.

### Summary of MVS/XA Facilities Available in Cross Memory Mode

The MVS/XA facilities available in cross memory mode can be divided into two categories: those services that are available to cross memory mode programs without restriction and those services that have special cross memory options or restrictions associated with their use. A list of the macro instructions available without restrictions to cross memory mode callers and a list of the macro instructions that have special options or restrictions for cross memory callers are provided in Volume 2 under the topic "Cross Memory Restrictions for Macro Instructions."

In addition to the services provided by the macro instructions, the following functions are available to cross memory mode programs without restriction:

Segment and page faults - The system function of resolving segment and page faults is supported for a unit of work executing in cross memory mode.

Dispatcher, interrupt handling - The MVS/XA dispatcher and interrupt handlers, except the SVC interrupt handler, support programs executing in cross memory mode, and these functions save and restore the additional status required by cross memory mode programs.

System tracing also traces cross memory information.

The macro instruction descriptions in Volume 2 give details about cross memory support.

## Cross Memory Structures

Cross memory uses a set of programming and data structures that can be divided into three functional areas: cross memory authorization, cross memory linkage, and linkage conventions.

"Cross Memory Authorization" describes how address spaces and programs are authorized to use PT, SSAR, and PC instructions and how the user can request that the system provide this authorization.

"Cross Memory Linkage" describes the structures and tables used by the PC instruction and how a user can request that the system create and connect these structures to particular address spaces.

"Linkage Conventions" describes a set of programming conventions that must be used to preserve register information and maintain system serviceability when using cross memory.

**Cross Memory Authorization:** Cross memory uses a more flexible authorization mechanism for inter-address space communication than the PSW key zero, supervisor state requirement for scheduling SRBs. There are multi-level authorization facilities that permit both supervisor and problem state programs in an address space to access programs and data in a selected set of address spaces, and also to restrict an individual problem state program's access to only a selected set of programs in other address spaces.

Programs have a selected set of PSW keys to which they are authorized, and, in problem state, this set controls the program's authority to access data in the secondary address space. Users request authorization by invoking a series of macro instructions.

**Address Space Authorization:** An address space's authorization to access other address spaces is based upon the authorization index (AX). Each address space has an AX. A program runs with the AX of the primary address space. The AX indicates the authority of the program to set another address space as its primary address space using the PT instruction and to set another address space as its secondary address space using the SSAR instruction.

The PT instruction is the mechanism used to return control from a PC routine (a routine that gets control as the result of a PC instruction). A program should use the PT instruction only to return to a program that called it using the PC instruction because instruction processing continues at the specified virtual address in the new primary address space, and system integrity and serviceability might be exposed by using the PT instruction in any other way.

Once a program has established another address space as its secondary address space, the program is authorized to move data between the secondary address space and the primary address space if the storage protection key of the data permits. The program can also directly reference data in the secondary address space by switching to secondary mode if the storage protection key of the data permits.

Each address space has associated with it an authorization table (AT). The AT contains one entry for every AX in use and indicates the authority of programs running with that AX to issue PT and SSAR instructions to the address space. The AX is used to index into the AT of the target address space on a PT or SSAR instruction to check if the issuing program has the authority to set the target address space as its primary or secondary address space. Authorization checking is required for both supervisor state and problem programs.

A particular address space, then, can selectively obtain PT and SSAR authority to a specific set of address spaces based on the ATs of those address spaces.

MVS/XA provides macro instructions to supervisor state or PKM 0-7 (the PKM is described below under "Program Authorization") programs to:

- Reserve an AX value for an address space (AXRES macro instruction)
- Free an AX value (AXFRE macro instruction)
- Set an address space's AX to a specified value (AXSET macro instruction)
- Set an address space's AT to indicate authorization levels for a specified AX value (ATSET macro instruction)
- Determine the AX value of an address space (AXEXT macro instruction)

All address spaces start with an AX of 0. An AX of 0 is an unauthorized AX value that prevents the address space from using PT and SSAR instructions. An AX of 1 is a fully authorized AX value that permits the address space to issue PT and SSAR instructions to any active address space. Certain system services that functionally serve all address spaces have an AX of 1. To have any other AX value, the user must explicitly reserve and set the AX with the AXRES and AXSET macro instructions, respectively. An address space to which this AX value is to be authorized (the address space to be accessed using PT and SSAR) must have its AT set using the ATSET macro instruction.

**Program Authorization - PKM (PSW Key Mask):** Each program has associated with it a PSW key mask (PKM) value. The PKM value can authorize individual programs to use cross memory. The PKM is a 16-bit string value that represents storage protection keys that are valid for the program to use, where bit n equal to 1 indicates that the program is authorized to use key n. The PKM is used only to perform authorization checking for problem state programs; supervisor state programs do not require PKM authority.

The PKM value is checked to see whether a problem state program can use the secondary access key specified on the MVCP and MVCS instructions to access storage in the secondary address space. It is also checked to see whether a problem program can use the secondary access key on the MVCK instruction.

The PKM value is also checked to see whether a problem program can issue a PC instruction. The PC instruction looks up an entry table entry (described later under "Cross Memory Linkage") that contains information for the PC instruction. Part of this information consists of an authorization key mask (AKM) value. The AKM is a 16-bit string value that indicates authorized keys in which a problem program can use a particular PC instruction. If the program's PKM indicates that it is authorized to use any of the keys indicated by the AKM, then the program can use the PC instruction. The PKM value also indicates whether a problem program can set a particular PSW key using the SPKA instruction.

All programs are initially dispatched with a PKM value equal to the bit mask representation of the field TCBPKF or SRBPKF. For example, X'0080' represents key 8 and X'8000' represents key 0. The PKM value can be changed using the PC and PT instructions and the MODESET SVC instruction.

The MODESET SVC sets the PKM value to the bit mask representation of the PSW key value when control returns to the program in problem state.

The entry table that contains information for the PC instruction also contains an execution key mask (EKM). The EKM is a 16-bit string value like the PKM and could contain additional keys to which the PC service is to be authorized. The EKM is ORed into the PKM when the PC routine receives control.

A program that issues a PT instruction specifies a 16-bit string value that indicates the PSW keys the program is authorized to use when the PT instruction is completed. This 16-bit string is ANDed with the original PSW key mask, and the result is placed in the PSW key mask in control register 3.

**Cross Memory Linkage:** Synchronous cross memory transfer of control is done with the program call (PC) instruction and the program transfer (PT) instruction. The PC instruction uses a PC number as input. The PC number is composed of two concatenated indexes, the linkage index (LX) and the entry index (EX). The PC instruction uses these indexes to perform a two level table lookup that causes a specific program to get control in the address space and mode specified in the table.

The first level table is the linkage table; the second level table is the entry table. Figure 22 shows how the PC number is used to access a particular entry table entry. The first portion of the PC number is the linkage index (LX), which selects a specific entry in the linkage table. The low order byte of the PC number is the entry index (EX), which is an index into the entry table pointed to by the linkage table entry. The entry table entry contains information that describes the program to receive control.

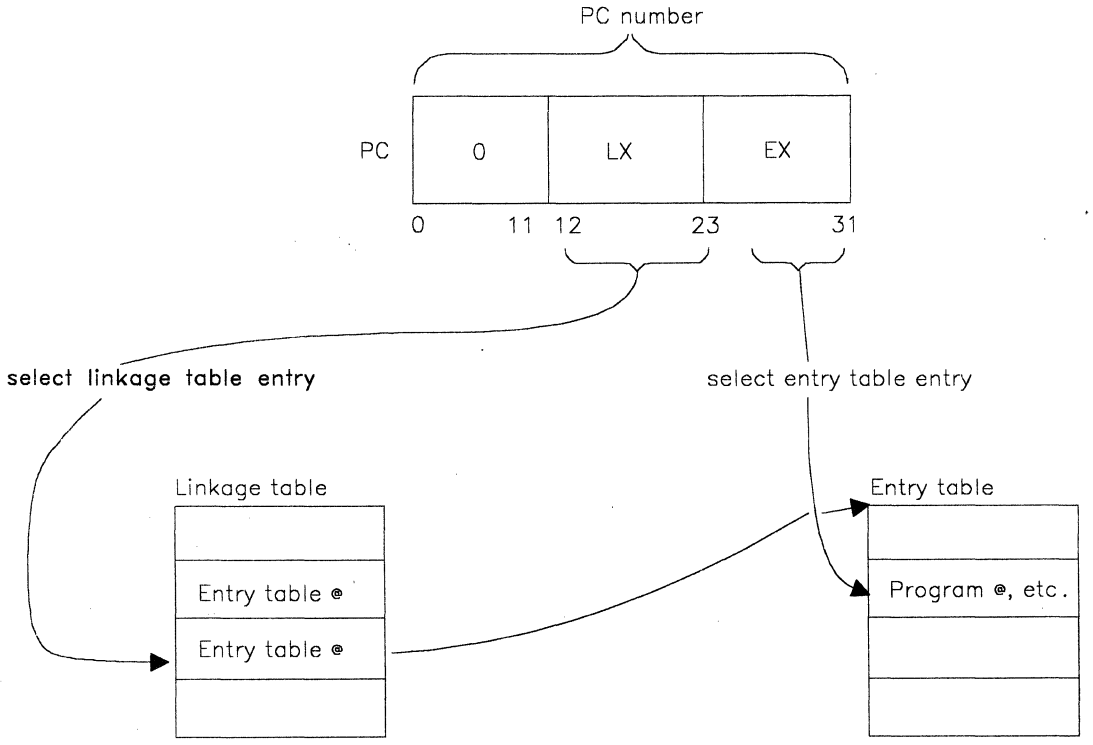



Figure 22. PC Number Indexing Linkage and Entry Tables

*Linkage Tables:* There is a linkage table associated with each address space in the system. An address space can have its own unique linkage table that gives it a set of cross memory services that is different from the set of services for any other address space. The linkage table can contain up to 1024 entries. Each linkage table entry can point to an entry table that describes a subset of the services available to the address space.

When a program wants to provide services via the PC instruction, it reserves an index into every linkage table in the system. The program connects an entry table to the reserved index for every linkage table whose address space is to have access to the services. To reserve an index, the program invokes the reserve linkage index service by issuing the LXRES macro instruction, which returns the reserved linkage index (LX). For further details, see "How to Establish a Cross Memory Environment" later in this section.

*Entry Tables:* Each program that provides services accessed via a PC instruction owns one or more entry tables. These entry tables are connected to the linkage tables of those address spaces that require access to the programs. Each entry in the entry table contains the following information about the program to be given control:

- Instruction address - specifies the addressing mode and virtual address in which the service is to receive control. (The addressing mode bit specifies the addressing mode of the called program as 24-bit or 31-bit.) For those entry table entries that do not describe user defined programs, this entry points to a special abend routine.
- ASID - specifies the ASID of the address space in which the called program will execute. If the value is zero, the program executes in the caller's primary address space.
- Problem state bit - specifies whether the called program will operate in problem state or supervisor state.
- Authorization key mask (AKM) - the AKM and the EKM are described earlier under "Cross Memory Authorization."
- Execution key mask (EKM) - the EKM and AKM are described earlier under "Cross Memory Authorization."
- Latent parameter address - specifies the address of a double word to be passed to the called program. The entry table creator supplies the first word. The second word is used by the PCLINK macro instruction. (PCLINK is described later in this section under "Linkage Conventions.")

An entry index (EX) is associated with each entry created in the entry table; the first entry has an EX of X'00' and subsequent entries have EXs of X'01' through X'FF'.

A program creates an entry table by issuing the ETCRE (create entry table) macro instruction, supplying all necessary details about the programs to receive control. These details go into the entry table. The ETCON and ETDIS macro instructions, respectively, connect and disconnect entry tables from linkage tables. The ETDES macro instruction destroys an entry table by removing it from the system. The uses of these macro instructions are described in greater detail in "How to Establish a Cross Memory Environment" later in this section. Figure 23 summarizes the macro instructions used to establish authorization and linkage.

| Macro Instructions For Authorization | Function |
|---|---|
| AXRES (Reserve AX) | Reserve authorization index |
| AXFRE (Free AX) | Return an AX for reuse |
| AXEXT (Extract AX) | Determine the AX of an address space |
| AXSET (Set AX) | Set the AX for an address space |
| ATSET (Set AT) | Set PT and SSAR authority in an authorization table entry |
| **Macro Instructions For Linkage** | |
| LXRES (Reserve LX) | Reserve a linkage index |
| LXFRE (Free LX) | Return an LX for reuse |
| ETCRE (Create ET) | Create an entry table |
| ETDES (Destroy ET) | Destroy an entry table |
| ETCON (Connect ET) | Connect an entry table to a linkage table at the specified LX |
| ETDIS (Disconnect ET) | Disconnect an entry table from a linkage table |

Figure 23. Authorization and Linkage Macro Instructions

**PC Numbers:** PC numbers are not permanently associated with a particular service the way SVC numbers are. The LX portion of the PC number is assigned by the control program and is not known before IPL. The EX portion is assigned by the component that owns a particular entry table. (While the component could make the EX portion of the PC number known by convention to the callers of its services, this is neither necessary nor desirable.)

Because the PC numbers themselves are not known before program execution, macro instructions and control program services cannot use PC numbers directly. Instead, PC numbers are determined indirectly by a table lookup process. For example, the PC numbers corresponding to many system functions are contained in a system function table (SFT) pointed to by the CVT. A macro instruction that invokes one of these PC services uses a permanently assigned index into the SFT to obtain the PC number for the service. A program that provides PC services must use a similar indirect method to give its callers the PC numbers they need to invoke its services. The caller of a service is not dependent on the actual PC number that is issued to obtain the service, on which module performs the service, or on where that module is located.

## Linkage Conventions

In a cross memory environment linkage conventions are more important than in other environments because the "how did I get here" information is essential. Therefore, users must save and restore status and diagnostic information in a consistent way for every program call/program transfer sequence.

When a program gets control as the result of a PC instruction, and uses PT to return, there are several things to be aware of:

- The called program must preserve registers 3 and 14 in order to return control with PT.
- The called program must preserve the PSW key and program mask across the PC/PT interface.

- If there is a dump when the called program is executing, the following information might be needed for the dump:

  Who called the currently executing program?
  What were the original contents of the caller's registers?
  Where is the caller's save area chain?

In order to preserve the above information, a program that is about to issue a PC instruction does the following:

- Saves registers 2 through 12 in the last 11 words (words 7 through 17) of a standard save area pointed to by register 13. You must save registers before issuing a PC because the PC instruction updates registers 3, 4, and 14, and the address space where the save area resides might no longer be the currently addressable address space.
- Saves the current SASID in bits 16-31 of save area word 5.
- Optionally loads registers 0, 1, and 15 as parameter registers.
- Loads register 2 with a PC number.
- Issues a PC specifying register 2.

The program that receives control as a result of the PC issues the PCLINK macro instruction with the STACK option to save linkage information. The PCLINK macro instruction can only be issued in supervisor state. The PCLINK macro instruction creates an area called a stack element (STKE), which contains the following information:

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

  In bits 0-23, bits 8-31 of caller's register 2
  In bits 24-27, PSW key
  In bits 28-31, zeroes

- Caller's PSW key mask and PASID from caller's register 3
- Latent parameter list address for this entry from caller's register 4
- Return address from the PCLINK service routine to the program that issued PCLINK STACK. This point is just after the PC routine entry point.
- Program mask from current PSW

After issuing PCLINK STACK, the program begins processing. It can, if it needs to, get information from the stack element using the PCLINK macro instruction with the EXTRACT option.

When the program is about to return control to its caller, it loads any data to be passed back to the caller into registers 0, 1, and 15 and then issues PCLINK with the UNSTACK,THRU option. This option restores registers 3, 13, 14, the program mask and, optionally, the original PSW protection key. The program then issues a PT instruction to return control. The caller restores its own registers and its SASID.

The PCLINK stack element is described in the *Debugging Handbook. Diagnostic Techniques* describes how the PCLINK stack elements and register save areas are chained together.

Figure 24 summarizes the PC/PT linkage conventions. In the figure, a program in ASID 8 issues a PC that invokes a program in ASID 7.

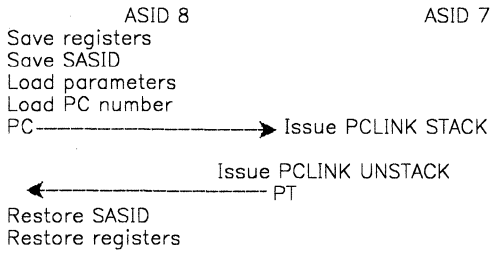



Figure 24. PC/PT Linkage Conventions

## How to Establish a Cross Memory Environment

This section contains three examples that show three ways to establish a set of services for access via a PC instruction. The term "subsystem" is used in this section but note that the functions providing cross memory services are not limited to those functions that use the subsystem interface. The required operations are grouped into five categories:

1. **SETTING UP** initializes the structure that cross memory needs so the transfers of control can take place.
2. **ESTABLISHING ACCESS** sets up the linkage necessary for an address space to use cross memory services.
3. **PROVIDING SERVICE** consists of designing a service for cross memory use. Refer to "Designing a PC Routine" later in this section.
4. **REMOVING ACCESS** disconnects the linkage that enabled an address space to use cross memory services.
5. **CLEANING UP** removes the structures established in the initialization step.

The first example shows how to make a cross memory subsystem's services available to a select group of users. (The code actually shows only one user, but the extra steps for adding users are pointed out.) The second example shows how to make a subsystem's services available system-wide to all address spaces. The third example shows how a subsystem can provide a series of non-space switch services that operate on data in the user's address space. A non-space switch service is one that does not cause an address space switch. See "Designing a PC Routine" later in this section.

Assume for all the examples that the subsystem has obtained common storage that can be accessed via the CVT. In this area it would store the PC numbers corresponding to its services. It could also store some of the lists that are needed to invoke PC/AUTH services, and that must be available to different address spaces. Assume also that SSBLOCK, shown in Figure 25, is in common storage accessible via the hypothetical CVT field, CVTXXXX. All examples use the declared storage areas shown in Figure 25.

```
SSBLOCK     DS  0D  SUBSYSTEM'S BLOCK
LXL         DS  0F  LX LIST
LXCOUNT     DS  F   NUMBER OF LXS REQUESTED
LXVALUE     DS  F   LX RETURNED BY LXRES
AXL         DS  0F  AX LIST
AXCOUNT     DS  H   NUMBER OF AXS REQUESTED
AXVALUE     DS  H   AX RETURNED BY AXRES
TKL         DS  0F  TOKEN LIST
TKCOUNT     DS  F   NUMBER OF ETS CREATED
TKVALUE     DS  F   TOKEN RETURNED BY ETCRE
PCTAB       DS  0F  TABLE OF PC NUMBERS
SERV1PC     DS  F   PC NUMBER FOR SERVICE 1
SERV2PC     DS  F   PC NUMBER FOR SERVICE 2
```

Figure 25. Declared Storage For Cross Memory Examples

## Example 1 - Making Services Available to Selected Address Spaces

**Setting Up:** To make its services available to other address spaces via a PC instruction, the subsystem sets up the linkage and entry tables and the authorization structures.

To request that the control program reserve an LX for later use, use the LXRES macro instruction to reserve a 4-byte LX across the entire system. When LXRES is issued, the home address space becomes the owner of the LX.

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

To set up the entry table describing the services and their entry points, use the ETCRE macro instruction. An entry table describes all the services the subsystem makes available to users through a PC instruction. The home address space, at the time the ETCRE macro instruction is issued, becomes the owner of the entry table.

First construct a list of entry table descriptors. Each descriptor, mapped by the IHAETD mapping macro instruction, describes a program that gets control when a PC is issued. Figure 26 shows an entry table descriptor list with two entries.

```
         .
         .
CET1     ETCRE   ENTRIES=ETDESC,RELATED=(CONET,DISET1,DESET2)
         ST      0,TKVALUE           SAVE RETURNED TOKEN
         .
```

```
ETDESC  DS 0D                ENTRY TABLE DESCRIPTION LIST
*                            (MAPPED BY IHAETD)
*------------------------ ETD HEADER
        DC X'00'             ETDFMT - MUST BE ZERO
        DC X'00'             ETDRSV1 - RESERVED, MUST BE ZERO
        DC H'2'              ETDNUM  - NUMBER OF ENTRY
*                            DESCRIPTIONS THAT FOLLOW
*------------------------ ENTRY 1
        DC X'00'             ETDEX - ENTRY INDEX (EX)
        DC X'C0'             ETDFLG - PROGRAM WILL EXECUTE
*                            SUPERVISOR STATE (ETDSUP ON)
*                            AND ENTRY WILL CAUSE SPACE
*                            SWITCH (ETDXM ON)
        DC H'0'              ETDRSV3 - RESERVED, MUST BE ZERO
        DC F'0'              ETDPR01
        DC A(SERVICE1)       ETDPR02 - VIRTUAL ADDRESS TO BE
*                            GIVEN CONTROL
        DC X'FFFF'           ETDAKM - CALLER CAN BE IN ANY KEY
        DC X'8000'           ETDEKM - SERVICE1 CAN ACCESS ONLY
*                            KEY 0 PLUS KEYS AUTHORIZED IN CALLER'S PKM
        DC F'0'              ETDPAR - LATENT PARAMETER
*                            PASSED TO CALLED PROGRAM
*------------------------ ENTRY 2
        DC X'01'             ETDEX
        DC X'80'             ETDFLG - PROGRAM WILL EXECUTE IN
*                            SUPERVISOR STATE (ETDSUP ON) AND
*                            ENTRY IN NON-SPACE SWITCH (ETDXM OFF)
        DC H'0'              ETDRSV3
        DC CL8'SERVICE2'     ETDPRO (PROGRAM NAME) PROGRAM
*                            MUST BE IN LPA
        DC X'00FF'           ETDAKM - CALLER MUST BE KEY 8-15
        DC X'00FF'           ETDEKM - SERVICE CAN ACCESS ONLY
*                            KEYS 8-15 PLUS KEYS AUTHORIZED IN CALLER'S
*                            PKM
        DC F'0'              ETDPAR - LATENT PARAMETER
*                            PASSED TO CALLED PROGRAM
```

**Note:** Upon entry, the PC routine receives a pointer, in general purpose register 4, to the latent parameter list. The first word of the latent parameter list is the value from the ETDPAR field.

Figure 26. Entry Table Descriptions for Examples

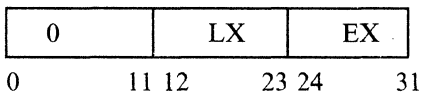
To request that the control program reserve an authorization index (AX) for the service, use the AXRES macro instruction. The AX is reserved across the entire system. The home address space at the time the AXRES macro instruction is issued becomes the owner of the AX.

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

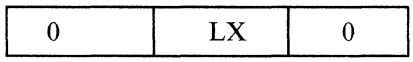
To set the AX of the subsystem's address space to the AX value the control program reserved, use the AXSET macro instruction.

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

At this point, you can construct the PC numbers that will be used to invoke the services. A PC number is a fullword value formed from an LX and an EX.




The linkage index returned by LXRES is in the following format so that you can OR it with an EX to form a PC number:




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

**Establishing Access:** The next two steps make the subsystem's services available to a user. The instructions used for these two steps must be issued from the user's address space but they must be invoked by a supervisor state or PKM 0-7 routine. If the user is a problem state program, the subsystem must arrange for the instructions to be executed on its behalf with the user's address space as the home address space. These two steps, of course, must be repeated for each user.

1. Set the PT and SSAR authority in the user's authorization table entry that corresponds to the subsystem's AX value so that the subsystem can issue a PT or SSAR instruction to the user's address space. This action allows the subsystem to access user data and return control to the user.

   ```
   SETAT  ATSET  AX=AXVALUE,PT=YES,SSAR=YES,RELATED=(GETAX,
                 SETAX,RESETAT)
   ```

2. Connect the subsystem's entry table to the user's linkage table at the entry that corresponds to the subsystem's LX. The linkage table now points to the subsystem's entry table.

   ```
            LA      2,1
            ST      2,TKCOUNT     SET COUNT OF ETS TO BE CONNECTED
   CONET    ETCON   TKLIST=TKL,LXLIST=LXL,RELATED=(GETLX,CET1)
   ```

Now all the user needs to get to the subsystem is the correct PC number. The subsystem devises a method of making its PC numbers available and makes the method known. The subsystem could use an executable macro instruction that expands into code that locates the PC number and then executes the PC instruction to invoke the desired service. The subsystem could keep the PC numbers in a table that each address space can locate in commonly addressable storage.

At this point in the example, the subsystem has set up two services that the user can access using PC instructions. The subsystem has also established authority to issue PT and SSAR instructions to the user. The user's linkage table is connected to the subsystem's entry table as shown in Figure 27.

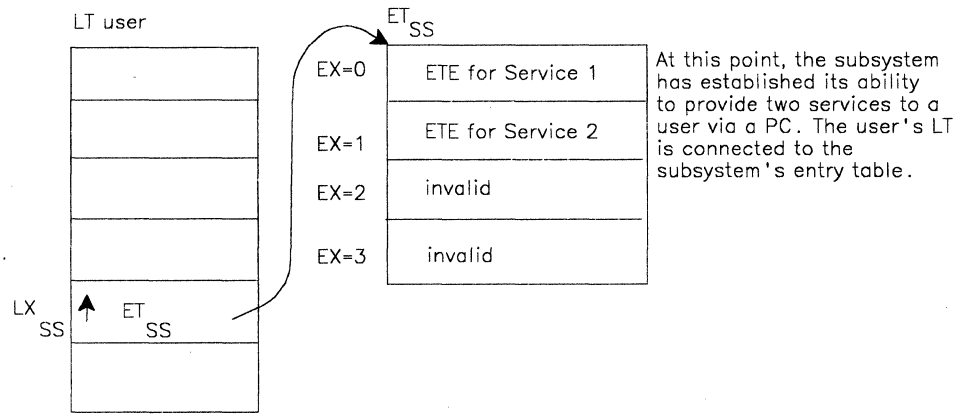



Figure 27. Linkage Table and Entry Table Connection

**Providing Service:** The PC instruction gives control to a PC routine that might run in cross memory mode. The PC routine must have been designed following the guidelines described in "Designing a PC Routine" later in this section.

To invoke the PC routine, the subsystem might provide a macro instruction. In the example, a macro instruction to invoke the first service would generate the following code:

```
STM  14,12,12(13)          SAVE REGISTERS
ESAR 2                     SAVE CALLER'S SASID IN THE
ST   2,16(,13)             REG 15 SLOT OF SAVEAREA
L    15,FLCCVT             ACCESS CVT
L    15,CVTXXXX(,15)       ACCESS SUBSYSTEM BLOCK
L    2,SERV1PC(,15)        OBTAIN SERVICE1 PC NUMBER
PC   0(2)                  ISSUE THE PC
L    14,12(,13)            RESTORE REG 14
L    2,16(,13)             LOAD SAVED SASID
SSAR 2                     RESTORE CALLER'S SASID
LM   2,12,28(13)           RESTORE REGS 2-12
```

**Removing Access:** The next two steps remove access to subsystem services. The steps are performed with the user's address space as the home address space. These steps are essentially the opposite of the steps used to establish access. First, remove the subsystem's PT and SSAR authority to the user's address space.

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

Second, disconnect the subsystem's entry table from the user's linkage table.

```
DISET1     ETDIS  TKLIST=TKL,RELATED=CET1
```

**Cleaning Up:** When the subsystem is about to shut down, it must remove all cross memory connections and release any cross memory resources it owns. Destroy the subsystem's entry table, first making sure that all connections to it have been disconnected.

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

Free the subsystem's linkage index so that another subsystem can reuse it.

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

Reset the subsystem's AX to zero.

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

Free the AX value so the system can reuse it. This action removes PT and SSAR authority corresponding to the subsystem's AX in all authorization tables in the system.

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

## Example 2 - Making Service Available to All Address Spaces

This example shows how a subsystem makes global services available system-wide to all users. The example uses the same storage areas as example 1, however, it does not need the AX list. Figure 25 and Figure 26 earlier, show the areas. The main differences between example 1 and example 2 are example 2's use of a system linkage index and a system AX value. There are only ten slots available in the system linkage table for the user to use. A system linkage index allows the subsystem to globally connect an entry table to all address spaces, and a system AX value gives the subsystem PT and SSAR authority to all address spaces.

**Setting Up:** The first step in the set up operation is obtaining a "system" linkage index. The control program sets aside part of the available linkage indexes for use as system LXs. When an entry table is connected to a system LX, the entry table is automatically connected to all present and future address spaces.

Unlike ordinary LXs, system LXs cannot be freed for reuse. When an address space that owns a system LX terminates, the LX becomes "dormant." The system allows a dormant system LX to be reconnected to an address space different from the original owning address space. This is an important consideration for a subsystem that can be terminated and then restarted. The subsystem must have a way to "remember" the system LX it owned so that it can connect the LX to an entry table when it is restarted.

There are two ways subsystems or components become owners of a system LX. Many IBM-supplied global services use a pre-assigned system LX and the PC numbers that correspond to their services are in the system function table (SFT).

The second way that a subsystem can get a system LX is by issuing the LXRES macro instruction with the SYSTEM = YES option.

The code shown in the following three steps runs with the subsystem's address space as the home address space. The first step obtains a system LX. If the subsystem is coming up for the first time since IPL and does not have a system LX preassigned in the SFT, issue the LXRES macro instruction with the SYSTEM=YES option. Save the LX somewhere, probably in common storage, so that it is accessible if the subsystem is restarted. On a subsystem restart, this step is not necessary.

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

Next, set the subsystem AX to 1. This allows the subsystem to issue a PT or SSAR instruction to all other address spaces because an AX of 1 is authorized to all address spaces. The subsystem that is providing a global service does not need to obtain a unique AX.

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

Define the subsystem's entry table as follows:

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

Next construct the PC numbers in the same way as in example 1.

**Establishing Access:** The following ETCON macro instruction, issued once from any address space, connects the subsystem's entry table to all address spaces in the system, current and future.

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

All address spaces in the system now have access to the subsystem's services. All linkage tables are connected to the subsystem's entry table and, because the subsystem's AX is 1, it can issue PT and SSAR to any address space. Figure 28 shows how the linkage and entry tables appear at this point.

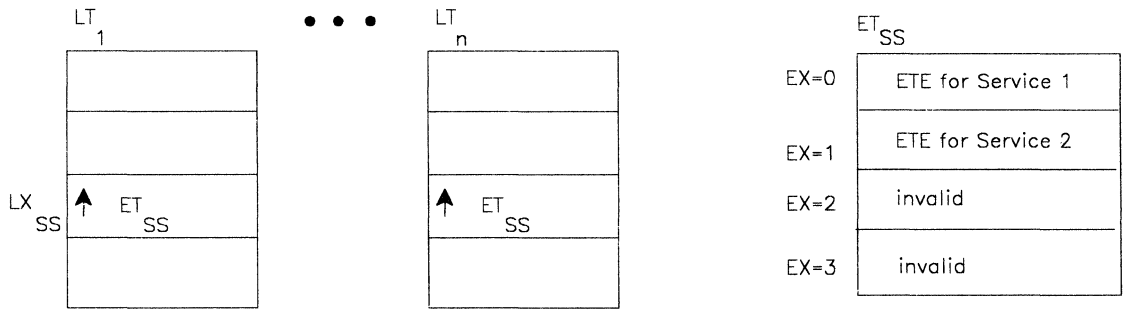



Figure 28. Linkage and Entry Tables for a Global Service

**Providing Service:** The subsystem provides service in the same way as in Example 1. The users of the services need to determine the PC number associated with each service. For many IBM-supplied services, the PC numbers are in the SFT. Other services must use a similar method.

**Removing Access:** To remove access, disconnect all users and destroy the entry table by issuing the ETDES macro instruction with the PURGE=YES option. This disconnects the entry table from all linkage tables in the system and then destroys it. (You cannot issue an ETDIS macro instruction for an entry table connected to a system LX.)

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

**Cleaning Up:** To clean up, reset the subsystem's AX to 0 as follows:

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

### Example 3 - Providing Non-Space Switch Services

In this example, a subsystem is providing a series of non-space switch services. Non-space switch services are described in "Designing a PC Routine" later in this section. This example is like example 1 except that the macro instructions for AX authorization are not used because an address space switch does not occur.

## Designing a PC Routine

PC routines get control as the result of a PC instruction. A PC routine serves the same general purpose as an SVC routine in that it is a means of providing a function at an increased level of authority. While the SVC always increases the authority of the SVC routine to key 0, supervisor state, the PC instruction allows greater flexibility in the authority that a PC routine can have. A PC instruction can switch to supervisor state, increase PKM authority, and switch to a different primary address space that has greater AX authority. The PC instruction can also prevent problem state programs from calling PC services based on PKM authority. For more information on the PKM and AX, see "Cross Memory Authorization" earlier in this section.

The SVC instruction requires that the issuer be in enabled task mode and hold no locks, but the PC instruction does not have these restrictions.

The PC instruction also allows the PC routine and the data it manipulates to reside in its own address space and be isolated from its callers.

The PC instruction must be issued in primary mode. The PC instruction can cause an address space switch. When you set up the entry table descriptor as input to the ETCRE macro instruction, one of the attributes specified is whether or not the PC routine is space switch.

When a non-space switch PC routine gets control, it executes in primary mode, and both the primary and secondary addresses are set to the address space in which the PC was issued.

When a space switch PC routine gets control, it executes in primary mode in the address space in which the entry table by which it is accessed was created, not the address space in which the PC instruction was issued. Secondary mode is set to the PC issuer's address space. Thus, the space switch PC routine gets control in cross memory mode.

All PC routines, both space switch and non-space switch, must:

- Be loaded under the job step task of the address space that created the entry table(s) that describe the routine or else be permanently resident (in PLPA or the nucleus).
- Preserve and restore PC linkage information.
- Use a PT instruction to return to their caller.
- Not use the checkpoint/restart facility.

PC routines that cause an address space switch have the following additional requirements:

- The address space in which the PC routine runs must be non-swappable.
- The PC routine must perform its functions using only the MVS/XA services supported in cross memory mode. Refer to "Warnings and Restrictions" earlier.

A space switch PC routine can access data in another address space by using the MVCP and MVCS instructions, or by using secondary mode. When executing in secondary mode, remember that all data is accessed from the secondary address space and that you cannot predict which address space (primary or secondary) instructions will be fetched from. Thus, all space switch PC routines that run in secondary mode must reside in common storage.

In deciding whether to make the PC routine a space switch or non-space switch routine, consider the nature of the routine itself and the data it manipulates. If either the PC routine or the data it manipulates needs to be isolated from the routine's callers, then the program or data, whichever requires isolation, should be located in the private area of an address space and the PC routine should be a space switch PC. If the program and data reside in commonly addressable storage or in the caller's address space, then the PC routine can be a non-space switch PC. A non-space switch PC can increase PKM authority and switch to supervisor state, but it cannot increase AX authority because no address space switch occurs.

Figure 29 shows the possible locations of the non-space switch PC routine and the data it manipulates and also lists the types of users who can invoke a non-space switch PC routine.

| **Location of PC routine:** | **Location of data to be manipulated:** | **PC routine can be connected to address space:** |
|---|---|---|
| Common | Common or caller's private area | All address spaces, via connection to a system LX or any address space via connection to an LX. |
| An entry table owner's private area | Common or caller's private area | Only entry table owner's address space |

Figure 29. Characteristics of a Non-Space Switch PC Routine

Figure 30 shows the possible locations of the space switch PC routine and the data it manipulates, what types of users can invoke the PC routine, and whether or not the routine can run in secondary mode.

| **Location of PC routine** | **Location of data** | **PC routine can be connected to:** | **PC routine can run in secondary** |
|---|---|---|---|
| Common | „ Common<br>„ Particular address space that owns ET<br>„ Associated data address space<br>„ Each calling address space | „ All address spaces with a system LX | Yes |
| Particular address space that owns an ET | | „ Any address space (to connect to a specific address space, the address space that creates the ET must be able to issue PT and SSAR instructions to the connected address space) | No |

Figure 30. Characteristics of a Space Switch PC Routine

## Recovery Considerations

There are special recovery considerations when you write a space switch PC routine. A PC routine executing in cross memory mode has active binds to address spaces other than home. If one or more of these address spaces terminates, then the PC routine will incur a program check and its recovery routine might get control. The SETFRR macro instruction provides options that specify the cross memory mode in which the recovery routine must get control. There are also options that enable a recovery routine to get control as a resource manager when the requested cross memory mode cannot be established in order to recover resources serialized by local (CML) or global locks. Refer to "Providing Recovery Routines" for details on recovery in cross memory mode. See "Locking" in the Serialization section for more information about the CML lock.

**Linkage Conventions:** The linkage conventions for PC and PT transfers of control have already been described under "Linkage Conventions" earlier. You will recall that the PCLINK macro instruction provides a standard method for saving status. The stack entries created by PCLINK are formatted like standard save areas so that you can trace the flow of control across address spaces in the event of a dump.

**Resource Management:** PC routines should be loaded under the job step task of the address space that created the associated entry tables. If the task under which the PC routine was loaded fails and it is not the job step task of the address space that created the entry tables, the PC routine is freed even though users are still connected to the inoperative PC routine, and results are unpredictable. If the PC routine is loaded under the job step task, any failure of the task causes a program check when any program issues a PC to the inoperative PC routine.

When a job step task that owns entry tables providing space switch PC services terminates, whether normally or abnormally, the space switch event mask for the address space is turned on. If this indicator is on, no unit of work can execute in cross memory mode in the address space. A unit of work currently using the space switch PC services or a unit of work attempting to issue a PC to the address space causes a space switch event program check. Subsequent job steps execute normally except that they cannot reestablish space switch PC services. If a unit of work in a subsequent job step attempts to reestablish space switch PC services (that is, issues an LXRES, AXRES, or ETCRE macro instruction), it causes a X'052' abend.

When a job step task that owns space switch entry tables terminates (normally or abnormally), the address space of the task is not terminated. The ASID representing the address space of the terminated task is retained and evaluated for possible reuse before the next IPL takes place. The reuse of address space spares an installation the burden of scheduling IPLs at frequent intervals to recover the lost space. The automatic recovery of used address spaces is an important consideration in the installation's choice of a MAXUSERS parameter. It is also a factor in how an installation controls the creation and termination of cross memory environments.

The system maintains a history of cross memory binds and address spaces. When all cross memory binds have terminated, address spaces that created space switch entry tables are generally reused, although there are a few special cases in which system integrity cannot be guaranteed if the spaces are reused. The system recognizes those cases (there are two) and prevents the reuse of the corresponding address space for the duration of the current IPL.

- The first case involves circular PC chains. For example, consider programs p1, p2, and p3 running in respective address spaces s1, s2, and s3. If PC instructions are issued by p1 to p2, and by p2 to p3, and then by p3 back to p1, a **circular PC chain** exists. Upon termination, address spaces such as s1-s3 that are used in a circular PC chain are considered non-reusable for the duration of the IPL.

- The second case involves any address space that has a cross memory connection to a **system** linkage index (LX). When this kind of address space terminates, it is considered non-reusable for the duration of the IPL. In addition, if this space is connected to any other address spaces, upon their termination they would also become non-reusable for the duration of the IPL.

A TCB for any job step task that owns a cross memory resource imposes a restriction on other TCBs that are higher up. The higher TCBs (that is, TCB for the initiator, RCT, DUMP, or STC) are restricted; they can only use **system** PC services. When the TCB that represents the task terminates, any connections between the higher TCBs and non-system entry tables are severed. Subsequent PCs that depend on those connections will not be successful.

While a PC routine is running, execution time is attributed to the home address space whether or not it is the same address space in which the PC routine executes.

# Virtual Storage Management

Virtual storage management (VSM) 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. In addition, VSM provides the following services through the use of the macro instructions specified:

- List the starting address and the size of the private area regions associated with a given TCB -- VSMREGN
- Verify that a given area has been allocated via a GETMAIN macro instruction -- VSMLOC
- List the ranges of virtual storage allocated in a specified area -- VSMLIST

These VSM 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. If you need to check whether a GETMAIN was issued to allocate a given block of storage, you can use the VSMLOC macro instruction 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 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 later in this topic under the heading "Obtaining Information about the Allocation of Virtual Storage."

# Allocating and Freeing Virtual Storage (GETMAIN, FREEMAIN)

The GETMAIN and FREEMAIN macro instructions respectively allocate and free one or more areas of virtual storage. The KEY parameter allows a user executing in PSW key zero to specify the storage key for storage requests involving subpools 227, 228, 229, 230, 231, and 241.

You can use the GETMAIN and FREEMAIN macro instructions when your program is executing in either 24-bit or 31-bit addressing mode. If you specify the options R, LC, LU, VC, VU, V, EC, EU, or E (provided by SVC 4, 5, and 10), storage addresses and lengths are treated as 24-bit addresses and lengths. If you want to specify 31-bit address and lengths, you must use the options RU, RC, VRC, or VRU. You can use the keyword LOC with these options to indicate the location of both virtual and real storage. See Figure 31 for a list of valid subpools and the location of these subpools when backed in real storage.

Most of the functions of GETMAIN and FREEMAIN (including the options mentioned above) are available to all users. However, some of the GETMAIN and FREEMAIN functions are available only to programs executing in supervisor state or PSW key zero. The restricted functions are provided by the parameters BRANCH and KEY.

## The BRANCH Parameter

In addition to the normal SVC entries to the GETMAIN and FREEMAIN macros, there are also branch entries, which are available through the BRANCH parameter. Although the branch entries require the user to do more work, they are more efficient than the SVC entries.

Branch entry to the GETMAIN or FREEMAIN macro instructions is accomplished by specifying BRANCH=YES on the macro instructions. If the BRANCH=YES parameter is used, the caller must preload register 4 with the TCB address, preload register 7 with the ASCB address, and hold the LOCAL lock. The contents of register 3 are destroyed if RC, RU, VRC, or VRU are specified with this parameter.

Callers in cross memory mode can use the BRANCH=YES parameter of the GETMAIN and FREEMAIN macro instructions. If the caller is in cross memory mode, the storage that is allocated or freed is located in the currently addressable address space. The caller must hold the CML lock for the currently addressable address space; load register 7 with the address of the ASCB of the currently addressable address space; and load register 4 with zero or the address of a TCB in the currently addressable address space. If register 4 contains a zero, the storage is associated with the current job step task that owns the cross memory resources in the currently addressable address space (that is, the TCB anchored in ASCBXTCB).

An additional branch entry point is provided to obtain global storage without the need for holding the LOCAL lock. This entry point is available to programs that contain no references to particular address spaces (for example, timer routines). The caller must be in key zero, supervisor state, and be disabled. In addition, the caller must hold no locks higher in the locking hierarchy than the VSMFIX lock for global subpools or the VSMPAG lock for subpools 231 and 241. Although the TCB address and ASCB address are not required for this entry, the macro expansion loads register 4 with the address of the global save area pointed to by the CVT.

Global branch entry can be obtained by coding BRANCH=(YES,GLOBAL) on the GETMAIN or FREEMAIN macro instruction that includes the positional parameter RC, RU, VRC, or VRU. The subpools that are supported by this entry are limited to the global subpools: common service area (CSA) subpools 227, 228, 231, and 241, and system queue area (SQA) subpools 226, 239, and 245. Any other subpool is considered an error.

### The KEY Parameter

The KEY parameter allows a user executing in PSW key zero to specify the storage key for storage he requests. Because branch entry users must be executing in PSW key zero at entry time, the KEY parameter satisfies the need to specify the actual key in which the requested storage is to be obtained.

The KEY parameter applies only to six subpools: 227, 228, 229, 230, 231, and 241. These subpools allow the requestor to obtain both global and local storage in key 0. (The KEY parameter allows an override of the PSW key.) Subpools 227 (fetch protected) and 228 (not fetch protected) are fixed global storage in the common service area, and must be freed explicitly. Subpools 229 (fetch protected) and 230 (not fetch protected) are local storage allocated from the top of the private area downward and intermixed with LSQA and SWA, and are freed automatically when the task terminates. Subpools 231 (fetch protected) and 241 (not fetch protected) are global storage in the common service area, and must be freed explicitly.

## Using Cell Pool Services (CPOOL)

The cell pool macro instruction provides users with another way of obtaining virtual storage. This macro instruction 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 instruction 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)

The CPOOL macro instruction, with the exception of the TCB, KEY, and LINKAGE=BRANCH parameters, is available to all users. 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:

- 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.

## Using Storage Subpools

Both the GETMAIN and the CPOOL instructions allow users to allocate storage from specified storage subpools.

The chart in Figure 31 lists the valid MVS/XA subpools and the characteristics of the subpools. It indicates the type of storage, whether the storage is fixed or fetch protected, where the storage is backed when fixed, and the storage key associated with the storage.

The storage map in Figure 32 shows the location of the storage areas listed in Figure 31. Virtual storage management allocates low private area storage beginning at the start of the private area or the start of the extended private area and it allocates high private area storage beginning at the upper end of the private area or the upper end of the extended private area.

The storage keys listed are:

| Key | Meaning |
|---|---|
| 0 | MVS/XA system control program |
| 1 | Job scheduler and job entry subsystem (JES2 or JES3) |
| USER | The storage key is taken from the PSW at the time of the GETMAIN or can be specified on the GETMAIN/FREEMAIN macro instructions. |
| JOB | The storage key is from the TCB associated with the request at the time of the first GETMAIN request. All subsequent GETMAIN requests use this key regardless of the key currently in the TCB. |

| Subpool | Type of Storage | Fixed | Fetch Protected | Where Backed | Storage Key |
|---|---|---|---|---|---|
| 0-127 | Low private | No | Yes | Below 16 Mb | Job |
| 226 | Common-SQA | Yes | No | Below 16 Mb | 0 |
| 227 | Common-CSA/ECSA | Yes | Yes | Below 16 Mb | User |
| 228 | Common-CSA/ECSA | Yes | No | Below 16 Mb | User |
| 229 | High Private | No | Yes | Below 16 Mb | User |
| 230 | High private | No | No | Below 16 Mb | User |
| 231 | Common-CSA/ESCSA | No | Yes | Below 16 Mb | User |
| 233 | Private-LSQA/ELSQA | Yes | No | Anywhere | 0 |
| 234 | Private-LSQA/ELSQA | Yes | No | Anywhere | 0 |
| 235 | Private-LSQA/ELSQA | Yes | No | Anywhere | 0 |
| 236 | High private | No | No | Anywhere | 1 |
| 237 | High private | No | No | Anywhere | 1 |
| 239 | Common-SQA/ESQA | Yes | Yes | Anywhere | 0 |
| 240 | Low private | No | Yes | Below 16 Mb | Job |
| 241 | Common-CSA/ECSA | No | No | Below 16 Mb | User |
| 245 | Common-SQA/ESQA | Yes | No | Anywhere | 0 |
| 250 | Low private | No | Yes | Below 16 Mb | Job |
| 251 | Low private | No | Yes | Below 16 Mb | Job |
| 252 | Low private | No | No | Below 16 Mb | 0 |
| 253 | Private-LSQA/ELSQA | Yes | No | Anywhere | 0 |
| 254 | Private-LSQA/ELSQA | Yes | No | Anywhere | 0 |
| 255 | Private-LSQA/ELSQA | Yes | No | Anywhere | 0 |

Figure 31. Characteristics of the Valid Storage Subpools

**Notes:**

1. All private area subpools are swappable. Common area subpools are not swappable.
2. All subpools allocated virtually in the extended area can be backed anywhere.
3. Subpools 0-127, 229, 230, 231, 240, 241, 250, 251, and 252 can be backed anywhere. However, if a page fix is requested for allocation in the nonextended areas, these subpools are backed below 16 megabytes real unless LOC is specified with ANY for real allocation.
4. Subpool 226 is valid only for allocating virtual storage below 16 megabytes.
5. Subpools 227 and 228 are backed anywhere for virtual addresses above 16 megabytes. For virtual addresses below 16 megabytes, they are backed below 16 megabytes unless the user of the GETMAIN instruction specifies the LOC parameter with ANY for real allocation.
6. Callers executing in key 0 and supervisor state, who request storage from subpool 0, via the GETMAIN macro instruction, obtain that storage from subpool 252. Therefore, if they want to dump the storage using the SDUMP macro instruction, they must specify subpool 252 rather than subpool 0.

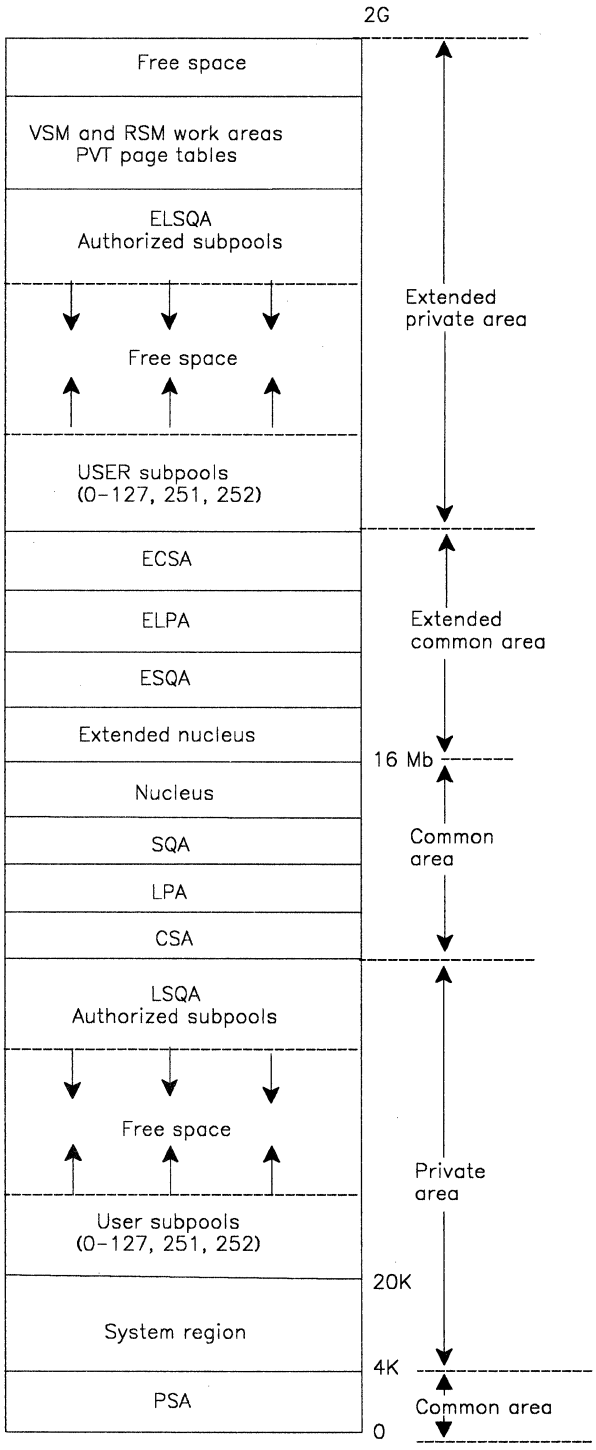



Figure 32. MVS/XA Virtual Storage Map

# Obtaining Information about the Allocation of Virtual Storage

The VSMLIST macro instruction 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 instruction. The length of the work area varies but it must be a minimum length of 4K bytes. Figure 33 shows the format of the VSMLIST work area.

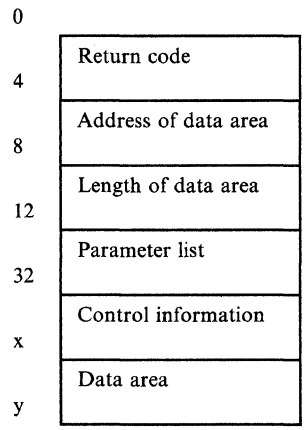



Figure 33. Format of the VSMLIST Work Area

## Using the VSMLIST Work Area

Prior to the first invocation of the VSMLIST macro instruction 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 instruction 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 instruction 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 instruction 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 instruction 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 instruction (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 instruction.

| Bytes | Field name | Description |
|---|---|---|
| 0-3 | Return code | This field contains the return code from the previous invocation of the VSMLIST macro instruction. You must set this field to zero before the first invocation of the VSMLIST macro instruction 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 instruction. |
| 12-15 | Parameter list | This section of the work area is constructed by the VSMLIST service routine according to the parameters that you specified when you issued the VSMLIST macro instruction.<br>**Bytes** **Contents**<br>12-15 Length of work area<br>16 SP operand represented as follows:<br>X'00' -- SQA<br>X'01' -- CSA<br>X'02' -- LSQA<br>X'03' -- PVT<br>X'FF' -- Subpool list provided<br>17 SPACE operand represented as follows:<br>X'00' -- ALLOC<br>X'01' -- FREE<br>X'02' -- UNALLOC<br>18 Information about the TCB, LOC, and REAL operands represented as follows:<br>X'80' -- ALL specified for the TCB operand<br>X'40' -- ANY specified for the LOC operand<br>X'20' -- REAL operand specified<br>19 Set to zero<br>20-23 TCB address or zero<br>24-27 Subpool list address or zero<br>28-31 Set to zero |
| 32-x | Control information | The control information is used by the VSMLIST service on multiple invocations for a single request. This area varies in size. |
| x-y | Data area | This area contains the actual output of the VSMLIST macro instruction. The area varies in size and is limited by the length of the work area specified as a parameter of the macro instruction. |

Figure 34. Description of VSMLIST Work Area

The information returned in the data area depends on the parameters specified on the macro invocation. You can use the VSMLIST macro instruction 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 via a GETMAIN macro instruction. Free space within that block is the area that has not been allocated via a GETMAIN macro instruction. An unallocated block of storage is some multiple of 4K none of which has been allocated via a GETMAIN macro instruction.

VSMLIST reports all SQA pages not allocated to subpools 226 and 239 as allocated to subpool 245. These pages of subpool 245 may not have been allocated via a GETMAIN 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 instruction. The format of the output varies according to what you specify for the SP parameter.

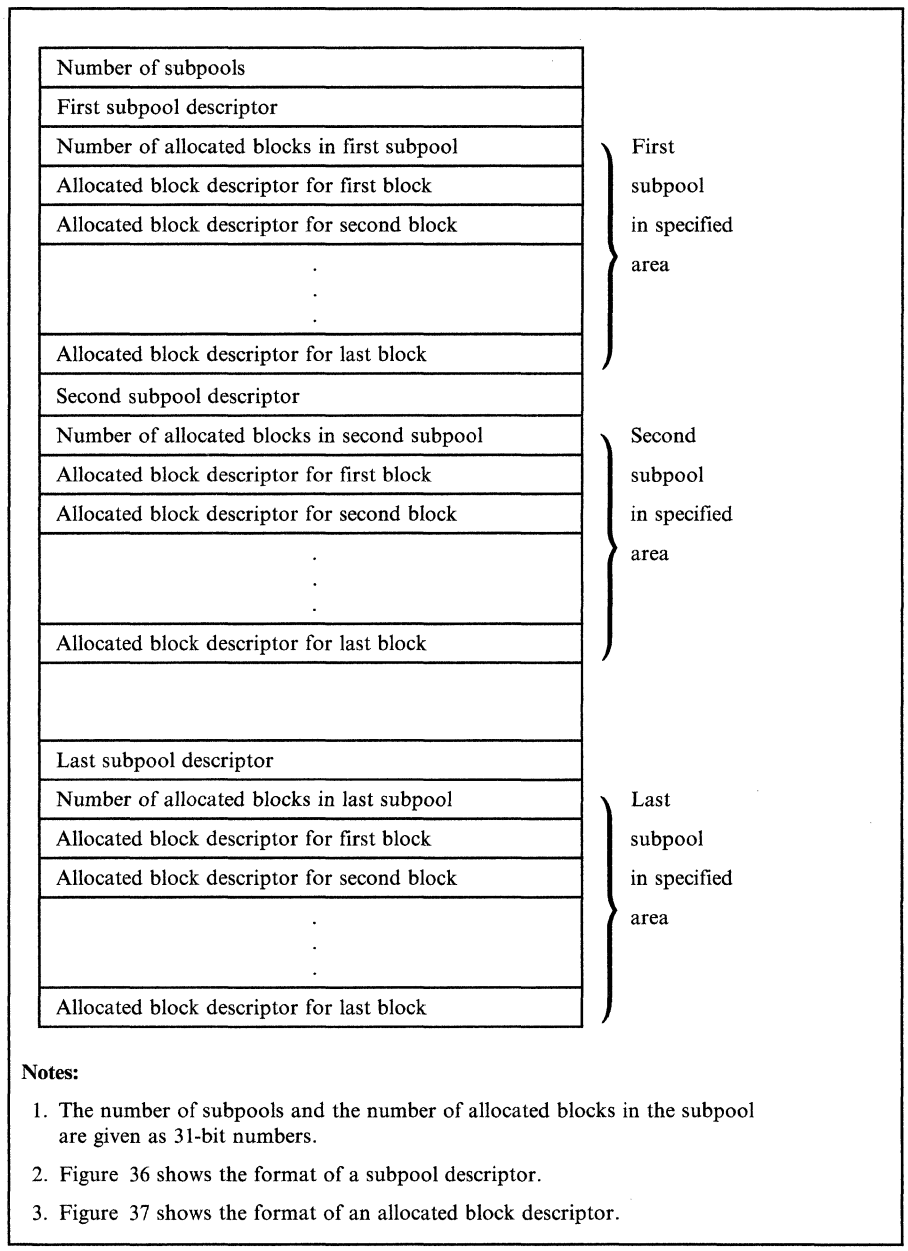
If you specify SP=SQA, SP=CSA, or SP=LSQA, the output consists of the allocated storage information for the subpools in the specified area. The subpools listed in each of these areas are:

SQA: 226, 239, 245
CSA: 227, 228, 231, 241
LSQA: 255

Figure 35 shows the format of the output for a request for information about the allocated storage in a specified area.

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, 229, 230, 236, 237, 251, and 252. Figure 38 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 39 shows the format of the allocated storage information for a subpool list request.




**Notes:**

1. The number of subpools and the number of allocated blocks in the subpool are given as 31-bit numbers.
2. Figure 36 shows the format of a subpool descriptor.
3. Figure 37 shows the format of an allocated block descriptor.

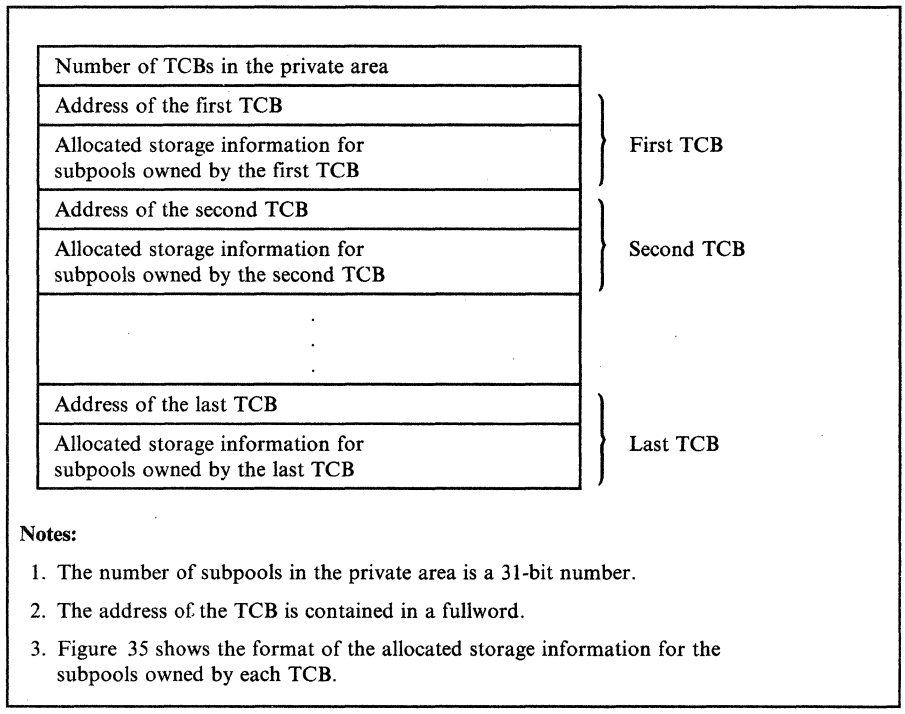
Figure 35. Allocated Storage Information for Subpools in a Specified Area

| 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<br><br>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 36. Format of Subpool Descriptor

| Byte | Content | |
|---|---|---|
| 0-3 | The virtual address of the allocated block | |
| | **Bit** | **Meaning When Set** |
| | 0 | The caller specified the REAL option and this allocated block could be backed in real storage above (bit 0 = 1) or below (bit 0 = 0) 16 megabytes. |
| 4-7 | The length of the allocated block | |

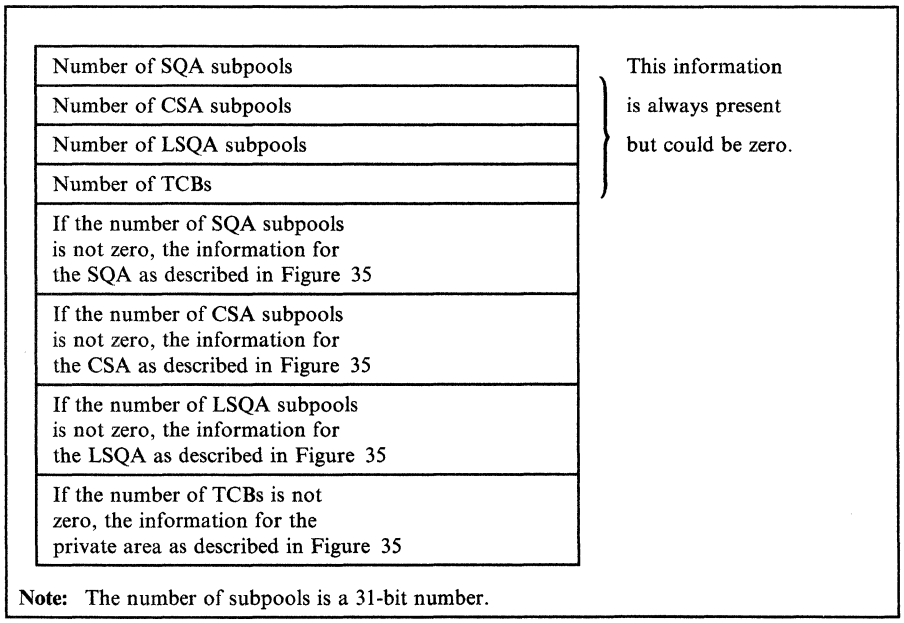
Figure 37. Format of Allocated Block Descriptor




**Notes:**

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 35 shows the format of the allocated storage information for the subpools owned by each TCB.

Figure 38. Allocated Storage Information for the Private Area




**Note:** The number of subpools is a 31-bit number.

Figure 39. Allocated Storage Information for a Subpool List

## Free Storage Information

A request for free storage information is specified by the SPACE=FREE parameter of the VSMLIST macro instruction. The VSMLIST service routine returns information about both allocated and free virtual storage. The information is returned in the same manner as allocated storage information except that each allocated block descriptor is followed by the number of pieces of contiguous free storage contained within the allocated block and the free space descriptors for each of these areas. Figure 40 shows the format of a free space descriptor.

| Byte | Content |
|---|---|
| 0-3 | The virtual address of the start of the free space |
| 4-7 | The length of the free space |

Figure 40. 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 instruction. You can obtain this information for CSA and private area subpools only, by specifying SP=CSA or SP=PVT. Figure 41 shows the format of the output for a SPACE=UNALLOC request for CSA or PVT subpools.

| | |
|---|---|
| Number of region descriptors | |
| First region descriptor | First region in specified area |
| 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 | Second region in specified area |
| 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 | Last region in specified area |
| Unallocated block descriptor for the first unallocated block | |
| Unallocated block descriptor for the second unallocated block | |
| . . . | |
| Unallocated block descriptor for the last unallocated block | |

**Notes:**

1. The number of region descriptors and the number of unallocated blocks in each region are given as 31- bit numbers.
2. Figure 42 shows the format of a region descriptor.
3. Figure 43 shows the format of an unallocated block descriptor.

Figure 41. 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 42. Format of Region Descriptor

| Byte | Content |
|---|---|
| 0-3 | The virtual address of the unallocated block |
| 4-7 | The length of the unallocated block |

Figure 43. Format of Unallocated Block Descriptor

## Accessing the Scheduler Work Area

When the system interprets JCL statements, it obtains information about jobs that are coming into the system. It stores this information in the scheduler work area (SWA). When jobs run, the system (dynamic allocation, for example) develops additional information about the jobs, which it also stores in the SWA. Some of this information is in the following SWA blocks:

- The job control table (JCT)
- The step control table (SCT)
- The account control table (ACT)
- 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 access the information in these blocks. Some of the blocks have accounting and timing information. Your program can use this information to generate reports of the system resources that your job uses.

For detailed information on coding the macros, see *SPL: System Macros and Facilities Volume 2*. By using these macros, you can read from a block, write into a block, or obtain the location of a block. You can also create or delete a SWA block, although creating and deleting a block requires special knowledge of the system. The only SWA blocks that you can access are the ones associated with your job.

As shown in Figure 44, SWA blocks have a prefix area and a data area:

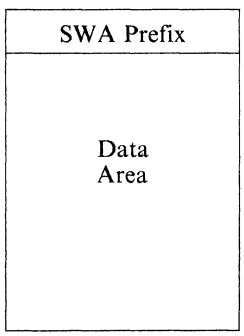



Figure 44. Format of a SWA Control Block

The part of an SWA block that you can access by using the SWAREQ and IEFQMREQ macros is the data area. Each macro has a different way of accessing a SWA block:

- 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 read or write information yourself.

## Using the IEFQMREQ and the SWAREQ Macros

To use these macros, you must be authorized, in task mode, and not in cross memory mode. However, when you are using SWAREQ to perform a Read Locate or a Locate All, you can override these restrictions by specifying the UNAUTH=YES parameter.

When you invoke the macros, you must provide a function code and a pointer to an external parameter area (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. Assign locate is the only function that requires you to input a block length.
- 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 listed in the IEFQMIDS 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 |
|---|---|
| JCT (job control table) | JSCBJCTA in the active JSCB |
| SCT (step control table) | JSCSCTP in the active JSCB |
| ACT (account control table) | JCTACTAD in the JCT or SCTAFACT in the SCT |
| JFCB (job file control block) | TIOEJFCB in the TIOT entry |
| JFCBX (job file control block extension) | JFCBEXAD in the JFCB |

## 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.

Other services (**Assign Locate**, **Assign Conditional**, **Delete Block**, and **Locate All**) are available but require special knowledge of the system:

**Assign Locate** — Obtains storage within the SWA for the type of block that you specify. Because the system has already assigned the necessary SWA blocks when your program executes, you would not normally use this service.

**Assign Conditional** — This service is the same as assign locate with the following exception: it does not abend if it cannot obtain the storage that it needs, but gives you a return code instead.

**Delete Block** — This service removes the block that you specify from the SWA. After this service executes, the specified block does not exist.

**Locate All** — This service returns the address of the data area and the address of the prefix area of the SWA block that you specify.

## How to invoke SWAREQ

As parameters of the SWAREQ macro, you specify the function code and the pointer to the EPA. The EPA input data and the EPA output data for each function code are summarized in the following block:

| SWAREQ Function | EPA Size | EPA Input Fields | EPA Output Fields |
|---|---|---|---|
| Assign Locate | 16 | Block length, block id | SVA, block pointer |
| Assign Conditional | 16 | Block length, block id | SVA, block pointer |
| Read Locate | 16 | SVA | Block pointer, block length, block id |
| Write Locate | 16 | SVA, block pointer, block id | None |
| Delete Block | 16 | SVA | None |
| Locate All | 28 | SVA, QMPA pointer | Block pointer, block id, block length, prefix pointer, prefix length. |

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 assembler needs definitions for the Communications Vector Table and the Job Entry Subsystem Communications Block. The following mapping macros provide the definitions that you need:

- IEFZB505 — EPA mapping macro
- IEFQMIDS — SWA block id definitions
- IEFJESCT — job entry subsystem communications block
- CVT — Communications vector table

When you specify UNAUTH=YES, you must observe the following rules:

- Your function code must specify Read Locate or Locate All.
- The EPA that you provide must be an extended EPA — an EPA that is 28 bytes long. To provide an extended EPA, use an option of the IEFZB505 mapping macro.
- If the job for which you are invoking SWAREQ is not the current job, or if the TCB of the job step is not addressable by PSATOLD, you must pass the 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 EPA that SWAREQ refers to by its input parameter.

## SWAREQ Summary

To issue a locate mode request, take the following steps:

1. Build an EPA (mapped by macro IEFZB505).
2. Issue the SWAREQ macro instruction, specifying the address of the EPA 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 register 6 points to the TIOT:

```
         LA     5,EPA                  GET ADDRESS OF THE EPA
         ST     5,SWEPAPTR             INITIALIZE EPA POINTER
         USING  SWAEPA,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
         IEFJESCT
         IEFZB505
         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 general register 15. If the return code is 0, the service is successful. Otherwise, the service failed and the non-zero return code in register 15 is also the reason code associated with the failure. In hexadecimal, the reason codes are:

08 — Invalid SVA in the SWA prefix
24 — Attempt to read a block not yet written
28 — Invalid pointer to the EPA

**UNAUTH=NO:** If you specify UNAUTH=NO or omit UNAUTH, the service can abend if an error occurs or if you are 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 general register 15. If the return code is 0, the service is successful. Otherwise, the service failed, and the non-zero return code in register 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 general registers 0, 1 and 15. Register 0 contains the address of an area that contains diagnostic information. Register 1 contains abend code 0B0. Register 15 has the reason code associated with the abend. The reason codes, in hexadecimal, are:

04 — Invalid function requested
08 — Invalid SVA in the SWA Prefix
0C — Attempt to read a block not yet written
10 — Invalid length for a SWA block
1C — Invalid block ID
20 — Invalid block pointer
24 — SVA does not correspond to any virtual address

## 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 IEFQMNGR mapping macro:

**Read** — Reads the data area of a specified block into your buffer.

**Write** — Writes information from your buffer into the block that you specify. Only the data area is written, not the prefix area.

Other services (Read All, Write All, Assign, and Write Assign) are available but require special knowledge of the system:

**Assign** — Obtains storage within the SWA for the type of block that you specify. Because the system has already assigned the necessary SWA blocks when your program executes, you would not normally use this service.

**Write Assign** — Writes information from your buffer into the block that you specify, and automatically assigns a section of SWA storage for another block. Only the data area is written, not the prefix area; the system fills in the prefix area. By the time your program executes, the system has already assigned the SWA blocks that are necessary for your job to run. Thus, you would not normally use the write assign.

**Read All** — Reads the block that you specify, including the data and the prefix areas, into your buffer.

**Write All** — Writes the data that is in your buffer into the block that you specify. The data and the prefix area are both written.

## 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 queue manager parameter area (QMPA) and the external parameter area (EPA). The input that you store in the QMPA is:

- **The function code** — The function code specifies the service to be performed.
- **The EPA address** — The EPA address, which can be in either of two QMPA fields, locates the EPA. Fill in the first field if you are specifying a three-byte address and the second field if you are specifying a four-byte address.
- **A 4-byte EPA address indicator** — Set this indicator if you are using a four-byte EPA address.
- **The extended EPA indicator** — Some services let you specify the size of the EPA as 8 or 16 bytes. Set this indicator if you are using the 16-byte EPA size.
- **The number of EPAs** — This is the number of times that the function is to be performed, and the number of EPA blocks that you are passing. 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.
- **The SWA manager subpool** — This field, which is necessary only for the Assign function code, specifies the number of the subpool that contains the SWA block to be assigned. The number must indicate subpool 236 or subpool 237.

If you want the function to be performed more than once, supply more than one EPA. 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 EPA, you must arrange them contiguously in storage. When you invoke the macro, general register 1 must point to the QMPA. The EPA input data and the EPA output data for each IEFQMREQ function code are summarized in the following block:

| IEFMQREQ Function | EPA size | EPA Output Fields | EPA Output Fields |
|---|---|---|---|
| Assign, Assign/Start | 4 | None | SVA |
| Assign, Assign/Start | 16 | Block id, block length | SVA |
| Read | 8 or 16 | SVA, buffer address | Block id |
| Write | 8 or 16 | SVA, buffer address, block id | None |
| Write/Assign | 8 | SVA for write, buffer address, write block id | SVA for Assign |
| Write/Assign | 16 | SVA for write, buffer address, write block id, length for assign, assign block id | SVA for Assign |
| Readall/Move | 8 or 16 | SVA, buffer address | Block id |
| Writeall/Move | 8 or 16 | SVA, buffer address, block id | None |

When you write a program that invokes IEFQMREQ, you must supply input data in fields that are in the QMPA and the EPA. You also need to supply SWA block definitions as input to the macro. When you assemble the program, the assembler needs definitions for the communications vector table and the job entry subsystem communications block. The format of the QMPA input data is defined in the *Debugging Handbook*. The format of the other input data is defined in the following mapping macros:

- IEFQMNGR — QMPA mapping macro
- IEFZB506 — EPA mapping macro
- IEFQMIDS — SWA block id definitions
- IEFJESCT — Job entry subsystem communications block
- CVT — communications vector table

## IEFQMREQ Summary

To issue a move mode request, take the following steps:

1. Build a QMPA (mapped by macro IEFQMNGR), which includes specifying the function code and setting a pointer to the EPA.
2. Build an EPA (mapped by macro IEFZB506).
3. Set register 1 to point to the QMPA.
4. Issue the IEFQMREQ macro.

## 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 register 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
         IEFJESCT
         IEFZB506
         IEFQMNGR
         IEFTIOT1
         IEFJFCBN
```

## Return Codes and Reason Codes from IEFQMREQ

When control returns after invoking IEFQMREQ, check the return code in general register 15. If the return code is 0, the service is successful. Otherwise, the service failed, and the non-zero return code in register 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 general registers 0, 1 and 15. Register 0 contains the address of the SDWA. Register 1 contains abend code 0B0. Register 15 has the reason code associated with the abend. The reason codes, in hexadecimal, are:

- 04 — Invalid function requested
- 08 — Invalid SVA in the SWA Prefix
- 0C — Attempt to read a block not yet written
- 10 — Invalid length for a SWA block
- 14 — Invalid count field
- 1C — Invalid block ID
- 24 — SVA does not correspond to any virtual address

# Real Storage Management

The real storage manager (RSM) administers the use of real storage and directs the movement of virtual storage pages between auxiliary storage slots and real storage frames in blocks of 4096 bytes. It makes all addressable virtual storage in each address space appear as real storage. Only the virtual pages necessary for program execution are kept in real storage. The remainder reside on auxiliary storage. RSM employs the auxiliary storage manager (ASM) to perform the actual paging I/O necessary to transfer pages in and out of real storage. ASM also provides DASD allocation and management for paging space on auxiliary storage. RSM relies on the system resource manager (SRM) for guidance in the performance of some of its operations.

RSM assigns real storage frames upon request from pools of available frames, thereby associating virtual addresses with real 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 storage frame, the page is considered pageable unless it is fixed by the FIX option of the PGSER macro instruction, a PGFIX or PGFIXA macro instruction, or obtained from a fixed subpool. RSM 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 real storage and is non-pageable.

The PGSER macro instruction in MVS/XA provides all the paging services through the use of parameters rather than separate macro instructions as in MVS/370. PGSER handles virtual addresses above or below 16 megabytes. The macro instructions, PGFIX, PGFIXA, PGFREE, PGFREEA, PGLOAD, PGANY, PGOUT, and PGRLSE are supported by MVS/XA to maintain compatibility with MVS/370, but it is recommended that you use the PGSER macro instruction.

Users should note that MVS/XA paging services function differently from MVS/370 paging services in the following ways:

- The end address (EA) specified on the PGSER macro instruction is the address of the last byte on which the page service is to be performed (not the last byte +1).
- In the register format SVC entry for the PGSER macro instruction, register 14 is used in addition to registers 0, 1, and 15.
- If an ECB is supplied, with a page-fix or page-load request and the caller invokes PGSER, then the return code must be checked because the ECB is not posted for a return code of 0.
- If an ECB is not supplied the return code need not be checked. Control will not be returned until the request is successfully completed. If the request fails, the caller will be abnormally terminated.
- Users of the PGSER macro instruction do not need to hold the local lock.
- Users of BRANCH=Y or BRANCH=SPECIAL options of the PGSER macro instruction must provide an 18-word savearea; this savearea must be in non-pageable storage if BRANCH=SPECIAL is specified.

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 real storage -- PGFREE, PGFREEA, or the FREE option of PGSER

- Fast path to free real storage -- the FREE and BRANCH=SPECIAL options of PGSER
- Load virtual storage areas into real storage -- PGLOAD or the LOAD option of PGSER
- Page out virtual storage areas from real storage -- PGOUT or the OUT option of PGSER
- Release virtual storage contents -- PGRLSE or the RELEASE option of PGSER
- Page anywhere (above or below the 16 megabyte (megabytes) line of real storage) -- PGANY or the ANYWHER 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 *Supervisor Services and Macro Instructions.*

## Fixing/Freeing Virtual Storage Contents

Fixing virtual storage and freeing real storage are complementary functions. The PGFIX and PGFIXA macro instructions and the FIX option of PGSER make specified storage areas resident in real storage and ineligible for page-out as long as the requesting address space remains in real storage. Note that page fixing ties up valuable real storage and is usually detrimental to system performance unless the use of the fixed pages is extremely high.

The PGFREE and PGFREEA macro instructions and the FREE option of PGSER make specified storage areas, which were previously fixed via the PGFIX macro instruction 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.

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 real storage and ineligible for page-out. These functions execute only short-term, synchronous page fixes.

**Notes:**

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 real 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. When using the PGFIXA macro instruction 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 instruction or the PGSER macro instruction with the FREE and BRANCH=SPECIAL options.

## 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 real 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 instructions 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 instruction 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.

See *Debugging Handbook* for an exact description of the VSL.

### 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.

See *Debugging Handbook* for an exact description of the PSL.

### Short Page Service List (SSL)

The short page service list provides the basic input to the PGSER macro instruction 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.

See *Debugging Handbook* for an exact description of the SSL.

## Branch Entry to the PGSER Routine

Branch entry to the PGSER macro instruction is available in both cross memory mode and non-cross memory mode for the FIX, FREE, OUT, LOAD, ANYWHER, and RELEASE options. The caller must be enabled, in supervisor state and key 0, and must set up the PSL as shown in *Debugging Handbook*. The caller does not need to hold the local lock, but must ensure that register 13 contains the address of an 18-word savearea when the PGSER macro instruction is issued.

The macro uses the registers as follows:

| Register(s) | Bit(s) | Contents |
|---|---|---|
| 0 | | ECB address or 0 if no ECB |
| 1 | 0 | 0 for register format 1 for list format |
| | 1-31 | Start of virtual area for register format<br>Pointer to the first PSL for list format |
| 2 | | 31-bit address of the last byte of the virtual area for register format<br>Irrelevant for list format |
| 3 | 0-15 | Reserved for register format |
| | 16-23 | Same as FUNC in PSL for register format |
| | 24-31 | Same as FLAG2 in PSL for register format<br>Irrelevant for list format |
| 4 | | TCB address or 0 for register format<br>Irrelevant for list format |
| 5-12 | | Not used |
| 13 | | Address of standard 72-byte save area, required for branch entry only.<br>For BRANCH = SPECIAL, the save area must be non-pageable. |
| 14 | | Pointer defined return address (The first bit indicates the AMODE. If this bit is 1, the AMODE is 31-bit; if this bit is 0, the AMODE is 24-bit. |
| 15 | | Entry point address |

On return from the PGSER macro instruction, the registers are set as follows:

| Register | Contents |
|---|---|
| 0-4 | The contents are destroyed and unpredictable. |
| 5-13 | The contents are unchanged. |
| 14 | The contents are destroyed and unpredictable. |
| 15 | Return code |

## Branch Entry to MVS/370 Page Services

Branch entry is available for all MVS/370 page services (page-fix, page-free, page-load, page-release, page-any, and page-out) in non-cross memory mode; and for all but the page-out service in cross memory mode. The caller must be in key 0, supervisor state, and must hold the local lock of the currently addressable address space.

**Note:** LOCAL lock means the local lock of the home address space. When written in lower case, the local lock refers to either the LOCAL or CML (cross memory local) lock.

### Cross Memory Mode

The pages that are candidates for page services must be addressable in the current address space. The caller must set up registers as follows:

| 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 (contents of CVTPSXM) |

On return, the page service sets the registers as follows:

| Register | Contents |
|---|---|
| 0-14 | Unchanged |
| 15 | Return code |

The only return code possible is 0. This indicates that the requested function was processed successfully.

**Note:** PGFIXA and PGFREEA can be invoked in cross memory mode.

## 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-anywhere service (contents of CVTVPSIB or PVTPSIB) |

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. |

# The Nucleus

The nucleus contains routines that execute with dynamic address translation (DAT) turned off and routines that execute with DAT on. These routines are located in two separate load modules. Load module IEAVNUC0n (n identifies the particular load module) contains the DAT-ON nucleus and load module IEAVEDAT contains the DAT-OFF nucleus. See *System Initialization Logic* for information concerning the manner in which the nucleus is loaded into storage.

There are two macro instructions that provide services for the nucleus. These macro instructions 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 |

## 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 instruction provides the means of linking to routines in the DAT-OFF nucleus.

When using the DATOFF macro instruction, 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 Figure 45.

| Index | Entry Point | Purpose |
|---|---|---|
| INDCDS | IEAVCDS | Compare Double and Swap |
| INDMVCL0 | IEAVMVC0 | General DAT-OFF MVCL function |
| INDMVCLK | IEAVMVKY | General DAT-OFF MVCL function in user key |
| INDXC0 | IEAVXC0 | General DAT-OFF XC function |
| INDUSR1 | IEAVEUR1 | User defined function |
| INDUSR2 | IEAVEUR2 | User defined function |
| INDUSR3 | IEAVEUR3 | User defined function |
| INDUSR4 | IEAVEUR4 | User defined function |

Figure 45. DAT-OFF Routines Available to Users

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 and must be capable of residing either above or below the 16 megabytes line. Therefore routines that execute in the DAT-OFF nucleus must have the attributes AMODE=31, RMODE=ANY. For information concerning 24-bit/31-bit compatibility, see *SPL: 31-Bit Addressing*.

### Using System Provided DAT-OFF Routines (DATOFF)

The system defined index values, INDMVCL0, INDMVCLK, INDXC0, and INDCDS 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 instruction destroys the contents of general registers 0, 14, and 15.

## INDMVCL0- Move Character Long

All register values must be 31-bit addresses. Before issuing the macro instruction, the user must load the registers as follows:

| Register | Use |
|---|---|
| 0 | Used by macro |
| 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 by macro |
| 15 | Used by macro |
| 1,6-13 | Unused |

The user invokes the MVCL function by coding the following macro instruction:

```
DATOFF INDMVCL0
```

## INDMVCLK- Move Character Long in User Key

All register values must be 31-bit addresses. Before issuing the macro instruction, the user must load the registers as follows:

| Register | Use |
|---|---|
| 0 | Used by macro |
| 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 by macro |
| 15 | Used by macro |
| 1,7-13 | Unused |

The user invokes the MVCL in user key function by coding the following macro instruction:

```
DATOFF INDMVCLK
```

## INDXC0 - Exclusive OR

All register values must be 31-bit addresses. Before issuing the macro instruction, the user must load the registers as follows:

| Register | Use |
|---|---|
| 0 | Used by macro |
| 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 orred with the operand pointed to by register 2. |
| 14 | Used by macro |
| 15 | Used by macro |
| 1,5-13 | Unused |

The user invokes the XC function by coding the following macro instruction:

```
DATOFF INDXC0
```

## IEAVMVC0- Compare Double and Swap

All register values must be 31-bit addresses. Before issuing the macro instruction, the user must load the registers as follows:

| Register | use |
|---|---|
| 0 | Used by macro |
| 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 by macro |
| 15 | Used by macro |

The user invokes the CDS function by coding the following macro instruction:

```
DATOFF IEAVCDS
```

## Writing User DAT-OFF Routines

As shown in Figure 45, there are four DAT-OFF indexes that users can define. These indexes are INDUSR1, INDUSR2, INDUSR3, and INDUSR4. The entry points corresponding to these indexes are IEAVEUR1, IEAVEUR2, IEAVEUR3, and IEAVEUR4, respectively.

User written DAT-OFF routines are restricted as follows:

- The user of the DATOFF macro instruction must be in key 0, supervisor state, and executing with DAT turned on.
- The DAT-OFF routine must have the attributes AMODE=31 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 routine must use BSM 0,14 to return.

See *SPL: System Modifications* for information about how user-written DAT-OFF routines are placed in the DAT-OFF nucleus.

# 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 46 shows the virtual storage map of the DAT-ON nucleus.

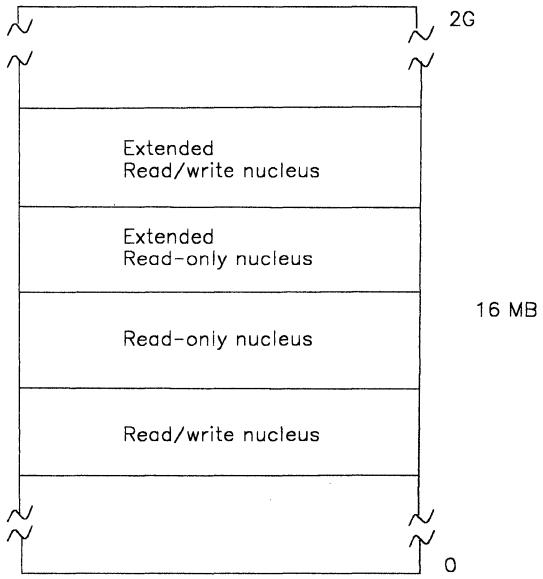



Figure 46. Virtual Storage Map of DAT-ON Nucleus

The nucleus map look up service provides users with information about these CSECTs. Through the use of the NUCLKUP macro instruction, 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.

# Normal and Abnormal Program Termination

The supervisor offers many services that help to detect and process abnormal conditions during system execution. The hardware detects certain types of abnormal conditions (such as an attempt to execute an instruction with an invalid operation code) and causes program interruptions to occur. The software detects other abnormal conditions (such as an attempt to open a data set that is not defined to the system) and causes abnormal terminations.

The supervisor enables you to write recovery routines to handle interruptions and abnormal conditions. The supervisor initiates the recovery termination process of your program either when you request it (for example, by issuing an ABEND macro instruction) or when MVS/XA detects a condition that will degrade the system or destroy data.

The services described in this section include:

- Invoking recovery termination (CALLRTM and ABEND macro instructions)
- Processing program interruptions (SPIE and ESPIE macro instructions)
- Intercepting system errors (SLIP command)
- Using system trace facilities (PTRACE)
- Dumping virtual storage (SDUMP macro instruction and CHNGDUMP command)
- Providing recovery routines (ESTAE, ATTACH with the ESTAI parameter, FESTAE, and SETFRR macro instructions)
- Uses of resource managers

## Recovery Termination Manager

The recovery termination manager (RTM) controls the flow of software recovery processing by handling all normal and abnormal terminations of tasks and address spaces. RTM gets control in response to events such as the following:

- Unanticipated program checks (except those protected by SPIE routines)
- Machine checks
- Invalid use of an SVC (issuing an SVC while locked, disabled, in SRB mode, or in cross memory mode)
- I/O error on page-in request
- ABEND or CALLRTM macro instruction requesting termination of a task or address space (see "Invoking the Recovery Termination Manager" later in this topic)

When one of these events occurs, RTM initiates recovery processing before proceeding with abnormal termination.

Your installation-written functions can use RTM's recovery processing by providing recovery routines for the functions. A recovery routine is a routine that you establish to get control if your main function terminates abnormally. The recovery routine can perform such processing as:

- Documenting the error
- Providing a dump of the storage needed to diagnose the error
- Freeing resources acquired by the main function

- Requesting a retry -- returning control to an appropriate point in the main function
- Requesting that RTM continue with the abnormal termination

To provide recovery for tasks and SRBs, RTM recognizes two types of recovery routines: functional recovery routines (FRRs) and ESTAE-type recovery routines. See "Uses of Resource Managers" later in this chapter for a full description.

When a function terminates abnormally, RTM gets control and generally invokes the most recently-established recovery routine to recover for the process that was in control. If this recovery routine cannot recover from the error (it fails or requests that termination continue), RTM invokes the next most recently established-recovery routine. The passing of control from one recovery routine to another is called *percolation.*

**Note:** MVS/XA 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 process in control (an SRB or a task) is terminated.

RTM invokes recovery routines only during abnormal termination of tasks or SRBs. RTM also invokes resource manager routines during both normal and abnormal termination of a task or an address space. The major purpose of a resource manager is to release any resources held by the task or address space and make these resources available to other users. See "Uses of Resource Managers" later in this chapter for a description of the processing such routines can perform.

## Invoking the Recovery Termination Manager

RTM can be called to perform its recovery and termination services on behalf of the caller or on behalf of another routine. Two macro instructions -- CALLRTM and ABEND -- invoke RTM.

### CALLRTM

A routine issues the CALLRTM macro instruction to direct the recovery termination services to a task or address space other than itself or its callers. Only key 0 supervisor state routines can issue CALLRTM. Control returns to the issuer of the macro instruction if TYPE=ABTERM or TYPE=MEMTERM is specified.

**TYPE=ABTERM:** If the CALLRTM macro instruction specifies TYPE=ABTERM, RTM processing is directed toward the specified task, and you should consider locking and work area requirements:

- If the TCB parameter is specified as 0 (or defaulted to 0) and the ASID parameter is omitted, the current task in the current address space is abnormally terminated. In this situation, the caller must be disabled (for example, hold any of the spin locks) and need not provide a work area via register 13. If dump options are supplied, they must be contained in fixed pages. The routine must exit to the dispatcher without changing the TCB or RB and without enabling.
- If the TCB parameter is specified as an address and the ASID parameter is omitted, the task associated with the specified TCB in the current address space is abnormally terminated. In this situation, the caller must own the LOCAL lock, and need not provide a work area. If the caller specifies a TCB equal to the current TCB address, the caller must also be disabled.
- If the ASID parameter is specified, the ABTERM function is scheduled as a service request block (SRB) to terminate the task in the specified address space. The caller, who specifies ASID, must pass the address of an 18-word save area in register 13.

**TYPE=MEMTERM:** If the CALLRTM macro instruction specifies TYPE=MEMTERM, RTM processing is directed toward an address space and you should consider the following locking and work area information:

- 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 an 18-word work area in register 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 an 18-word work area in register 13.

  **Note:** The required work area is not the standard 18-word save area; therefore, standard IBM linkage conventions do not apply to it. One aspect of this difference is that CALLRTM does not save registers in this work area in the same order as it would in a standard save area. All 18 words are used.

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. These routines include:

- Paging supervisor, when it determines that it cannot swap in the LSQA for an address space
- Memory create, when it determines that an address space cannot be initialized
- RTM or supervisor control functional recovery routine (FRR), when it determines that uncorrectable translation errors are occurring in the address space
- RTM, when it determines that task recovery and termination cannot take place in the current address space
- Region control task, when it has determined that the address space might become permanently deadlocked -- that is, unusable -- or that the status of the address space is unpredictable because of an error during swap-out processing
- RTM, when all tasks in the address space have terminated
- Auxiliary storage management (ASM) recovery, when it has an indeterminate error from which it cannot recover while handling a request for either swap-in or swap-out
- SVC 34, in response to a FORCE command

In addition, the terminal control address space (TCAS) specifies TYPE=MEMTERM when the system operator replies "FSTOP" (forced stop) to certain messages that can occur when TSO/VTAM time sharing starts or stops. The messages are IKT001D (replying "FSTOP" cancels terminal users already active when TSO/VTAM is starting) and IKT010D (replying "FSTOP" cancels terminal users still active when TSO/VTAM is being stopped). In both cases, the system operator should reply "FSTOP" to cancel users only if "SIC" (system-initiated cancellation) is ineffective. Replying "SIC" does not cause task resource manager processing to be bypassed.

## ABEND

Any routine, including supervisor state, locked, disabled, or SRB routines, can issue the ABEND macro instruction 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 abended or to enter recovery. Control never returns to the issuer of the macro (except as a result of a retry). See *Supervisor Services and Macro Instructions* for a description of the ABEND macro instruction.

# Processing Program Interruptions (SPIE, ESPIE)

The SPIE macro instruction 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 instruction 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 instruction.

Each succeeding SPIE/ESPIE macro instruction completely overrides any previous SPIE/ESPIE macro instruction specifications for the task. The specified exit routine gets control in the key of the TCB (TCBPKF) when one of the specified program interruptions occurs in any problem program of the task. When a SPIE macro instruction 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 instruction from within a SPIE exit routine, it has no effect on the contents of the PIE. However, if an ESPIE macro instruction 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 control program 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 instruction can be issued by any problem program being executed in the performance of the task. The control program automatically deletes the SPIE exit routine when the request block (RB) that created the SPIE macro instruction terminates.

The SPIE and ESPIE macro instructions and their related services are discussed in detail in *Supervisor Services and Macro Instructions*. The syntax of both the SPIE and the ESPIE macro instructions appears in Volume 2.

## Interruption Types

The programmer can specify interruptions 1-15 using either the SPIE or the ESPIE macro instruction. The installation-authorized system programmer can also specify interruption 17. 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 (TCBPKF) when a page fault occurs for the program that issued the SPIE/ESPIE macro instruction. The exit routine gets control in the addressing mode that was in effect when the SPIE or ESPIE macro instruction was issued. (If a SPIE macro instruction 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.

A caller in supervisor state, who issues the SPIE macro instruction is abnormally terminated with a 30E abend completion code. A caller in supervisor state, who issues the ESPIE macro instruction 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 instruction 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/ESPIE specifying interruption type 17 is in effect, the program check first level interrupt handler (FLIH) passes control to a SPIE/ESPIE service routine, which then passes control to the SPIE/ESPIE exit routine via an LPSW. The SPIE/ESPIE service routine sets up functional recovery routines (FRRs) to handle possible page faults caused by PIE/PICA references (for a SPIE) or EPIE references (for an ESPIE) during this set-up processing. If such a page fault occurs, the SPIE/ESPIE service routine returns to the FLIH, which invokes the paging supervisor to handle the original page fault. In this case the SPIE/ESPIE exit routine does not handle the original page fault because the SPIE/ESPIE service routine 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 instruction), the EPIE (for an ESPIE macro instruction), the SPIE/ESPIE program, and data areas in fixed storage. Page faults can occur after issuing the SPIE/ESPIE macro instruction 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.

**Note:** In MVS/370 the SPIE environment existed for the life of the task. In MVS/XA the SPIE environment is deleted when the request block containing the macro is deleted. That is, when a program running under MVS/XA completes, any SPIE environments created by the program are deleted. This might create an incompatibility with MVS/370 for programs that depend on the SPIE environment remaining in effect for the life of the task rather than the request block.

## Intercepting System Errors

Intercepting system events provides a way to get information about software conditions, in addition to the information normally supplied by dumping services during abnormal termination. The intercepting process, known as serviceability level indication processing (SLIP) is a major debugging tool.

SLIP definitions, called traps, specify the system conditions at the time of interception and the action that is to be taken after the interception. There are two types of SLIP traps: non-PER traps and PER traps. A non-PER trap obtains information about an error condition normally handled by the recovery/termination manager (RTM), such as a program check, abend, or restart interrupt. A PER trap uses program event recording (PER) hardware to obtain information about any of the following PER events:

- Successful branch: Successful execution of a branch taken within a user defined range of virtual addresses.
- Storage alteration: Alteration of the contents of a virtual storage location within a user-defined range of virtual addresses.
- Instruction fetch: Fetching and execution of an instruction within a user-defined range of virtual addresses.

Once the trap is defined and enabled, SLIP processing checks current system conditions for each SLIP event. PER traps are checked each time a PER event occurs; non-PER traps are checked when RTM processes an error. When current system conditions match the conditions specified in the trap, the action specified in the trap occurs. Normally, the action specified,

such as scheduling an SVC dump or writing a GTF trace record, is designed to collect diagnostic data.

SLIP users can minimize the effect of PER traps on system performance by limiting the scope of a SLIP trap to, for example, a particular address space or job. A SLIP trap can also include controls that automatically disable the trap when it is causing excessive overhead or has collected the required amount of data. The action to be taken when the trap matches can also be tailored. For example, SLIP users can tailor the SVC dump to include only diagnostic material essential for debugging. In addition, SLIP users can specify the type and contents of the GTF trace records. For more information on the SLIP command operands, see *Operations: System Commands*, *TSO/E Programming Guide*, and *TSO/E Programming Services*.

## Using the SLIP Command

Use of the SLIP command should be restricted to system programmers.

The command uses three operands to control SLIP traps:

- SET -- establish SLIP traps
- MOD -- modify SLIP traps
- DEL -- delete SLIP traps

It is also possible to display information about SLIP traps by using the DISPLAY command at the operator's console or from a TSO terminal. For specific information about how to enter both the SLIP and DISPLAY commands, refer to *Operations: System Commands* or *TSO/E System Programming Command Reference*. The descriptions in those publications also explain the operands. For more information on designing effective SLIP traps, see *Diagnostic Techniques*.

## Obtaining an SVC Dump During Slip Processing

SLIP processing invokes SDUMP (if A=SVCD or A=TRDUMP is specified) to obtain an SVC dump at the time the SLIP trap is entered. It uses SDURGPSA (a system parameter of the SDUMP macro instruction, not available for general use) to specify the following information:

- The PSW at the time of the error
- The contents of control register 3 and 4
- The contents of general registers 0-15

SVC dump puts this information into the dump header record. Summary dump dumps 4K of storage (for one address space only) around the PSW at the time of the error. The address space used depends on the setting of the S-bit. (If the S-bit is on, it dumps this area in the secondary address space; if off, it dumps this area in the primary address space. In addition, it dumps 4K of storage around the addresses in all general registers for both address spaces.

SDUMP copies the storage specified by the SDURGPSA parameter into SDUMP storage before returning to SLIP. This allows SLIP to continue without waiting for SDUMP to complete.

## Bypassing Dump Suppression

Dump analysis and elimination (DAE) suppresses duplicate SDUMPs and SYSMDUMPs under certain conditions. However, you can use the ACTION keyword of the SLIP command to override DAE. When all of the match conditions are met for a non-PER SLIP trap, the ACTION=NOSUP keyword prevents suppression of a recovery routine's dump request.

The SLIP actions of SVCDUMP and TRDUMP (trace dump) also override DAE. These actions trap critical problems that always require a dump.

"Suppressing SDUMPs and SYSMDUMPs" later in this section provides more information about DAE and contains a list of other publications that describe DAE.

## System Trace Facilities

MVS/XA system trace provides a record in storage of significant software events. System tracing is in effect, by default, in MVS/XA. The system trace facility consists of branch tracing, address space tracing, and explicit tracing. Each of these types of tracing causes entries to be made in the trace table. The trace table consists of one queue of trace buffers for each processor. These buffers are located in the private area of the trace address space. *Diagnostic Techniques* provides diagnostic information about the system trace table.

### Performing Branch Tracing

Branch tracing causes entries to be made in the trace table when the following branch instructions execute successfully:

- Branch and link register (BALR)
- Branch and save register (BASR)
- Branch and save and set mode (BASSM)

### Performing Address Space Tracing

Address space tracing causes entries to be made in the trace table when the following instructions execute successfully:

- Program call (PC)
- Program transfer (PT)
- Set secondary ASID (SSAR)

### Performing Explicit Tracing (PTRACE)

System services use the PTRACE macro instruction to explicitly trace normal system events. Callers in key 0 and supervisor state can also use the PTRACE macro instruction to make explicit entries in the trace table. These entries consist of an event identifier, the contents of a designated range of general registers or storage locations and system supplied status information. See the *Debugging Handbook* for examples of trace records.

The TRACE operator command dynamically controls tracing. It allows the operator to start or stop tracing, set the trace table size, and select the type of tracing. The TRACE command is described in *Operations: System Commands.*

# Dumping Virtual Storage

There are a variety of ways in which users can obtain dumps of virtual storage. The macro instructions that produce dumps are ABEND, SNAP, and SDUMP. Any user can issue the ABEND or SNAP macro instruction to request a dump of virtual storage. These unprivileged macro instructions are described in *Supervisor Services and Macro Instructions*. A system routine that encounters an error can use the SDUMP macro instruction to obtain a dump. The syntax of the SDUMP macro instruction is given in Volume 2 and a description of its use follows.

There are several operator commands related to controlling, taking, displaying, and suppressing dumps. These commands are: CHNGDUMP, DISPLAY DUMP, DUMP, DUMPDS, and SET DAE. The syntax and description of these operator commands are given in *Operations: System Commands*. A brief description of the way in which a system programmer can use these commands is given later under the heading "Using the Dumping Services Commands."

Additional information on dumps, of interest to system programmers, can be found in *SPL: System Modifications, Diagnostic Techniques, Debugging Handbook, IPCS User's Guide and Reference*, and *Service Aids*. *SPL: System Modifications* describes pre-dump suppression, post-dump installation exits, and dump data sets. *Diagnostic Techniques* provides information on the analysis and use of dumps. *Debugging Handbook* describes the types of dumps and also gives samples of the output from dumps. *Service Aids* describes Print Dump (PRDMP), including the improved and reformatted summary dump output that it produces in MVS/XA. *IPCS User's Guide and Reference* describes the various subcommands that allow you to analyze a dump online.

This topic contains information that a system programmer needs to control and use MVS/XA dumping facilities. It includes the following subtopics:

- Using the IPCS Macro Instructions
- Using the SDUMP macro instruction
- Obtaining an SVC dump
- Obtaining a summary dump
- Using dump data sets
- Using the dumping services commands
- Cancelling and restarting the DUMPSRV address space
- Getting more than one SYSMDUMP
- Suppressing SDUMPs and SYSMDUMPs

## Using the IPCS Macro Instructions

Interactive Problem Control System (IPCS) provides several macro instructions that PRDMP, SNAP, IPCS and user-written exit routines can use to tailor dump output.

The BLSQMDEF and BLSQMFLD macro instructions are used together to create a control block model. The format model processor service and the control block formatter service under IPCS and PRDMP use this model to format a control block in a dump. The model is a read-only structure that resides in a load library or in a CSECT within an exit program, and describes the control block to the formatter. The model consists of a header, defined by the BLSQMDEF macro instruction, and an array of entries, defined by the BLSQMFLD macro instruction, that describe individual fields.

The BLSRESSY macro instruction maps the IPCS symbol table record for use in the get symbol and equate symbol services. With the BLSRESSY macro instruction, users of the get symbol and equate symbol services can retrieve definitions described in the table and can create definitions for later use by the IPCS user or by other routines.

The BLSABDPL macro instruction maps the exit parameter list (BLSABDPL), a data area that enables IPCS, PRDMP, SNAP, and user-written exit routines to tailor dumps. With the BLSABDPL macro instruction, users can access different parameter lists within the BLSABDPL parameter list and then invoke the corresponding exit service routine.

For the syntax of the macro instructions, see *SPL: System Macros and Facilities Volume 2*. For information about the exit service routines, see *IPCS Planning and Customization*.

## Using the SDUMP Macro Instruction

System routines use the SDUMP macro instruction to obtain fast unformatted dumps of virtual storage. The SDUMP macro instruction invokes SVC dump to provide these services. Only one SVC dump can be taken in the system at one time.

Users can issue the SDUMP macro instruction while executing in either 24-bit or 31-bit addressing mode. The routine that issues the macro instruction regains control in the same addressing mode it was in when it requested the dump. SVC dump can handle both 24-bit and 31-bit addresses. It uses the value of the PSW, at the time of the error, to determine the addressing mode of the storage to be dumped. Users who switch addressing mode should note that SVC dump interprets the contents of registers and the addresses of parameter lists according to the caller's addressing mode at the time of the error.

### Using SDUMP in a Reentrant Program

Callers who want to generate reentrant code must code the list and execute forms of the SDUMP macro instruction. In order to use the list and execute form of the SDUMP macro instruction, the caller needs to know the length of the parameter list that is generated by the list form of the SDUMP macro instruction. The length of the parameter list varies depending on the options that you specify.

The SDUMP parameter list is a minimum of 40 bytes in length. The following conditions will change the size of the parameter list:

- If any of the following options are specified, the parameter list is at least 48 bytes long:

  TYPE=XMEM or XMEME
  SDATA=GRSQ
  LISTA
  SUMLSTA
  SUSPEND=YES or NO

  If any of the following options are specified, the parameter list is at least 68 bytes long:

  TYPE=NOLOCAL
  SUBPLST
  KEYLIST
  SDATA=ALLNUC

- If any of the following options are specified, the parameter list is at least 128 bytes long.

  PLISTVER=2
  SYMREC
  ID
  IDAD
  PSWREGS
  SDATA=DEFAULTS
  SDATA=NODEFAULTS
  SDATA=IO

If the HDR, STORAGE, or ID option is specified, the length of these areas must be added to the value 40, 48, 68, or 128, as indicated above, to obtain the total length of the parameter list.

A caller can determine the amount of storage needed for the parameter list of the SDUMP macro instruction in the following ways:

- Include the IHASDUMP mapping macro in the program, using the macro variable %SDUMP_PLISTVER='2'. This provides the mapping of all of the parameters (whether specified or not) and is the maximum length, excluding the length of the HDR, STORAGE, and ID options.
- Dynamically compute the amount of storage needed and place the result in SDUMPLEN:

```
SDUMPBEG SDUMP SDATA=(SUM),SUBLIST=SLIST,MF=L
SDUMPEND EQU   *
SDUMPLEN DC    A(SDUMPEND-SDUMPBEG)
```

## SDUMP Options

The parameters of the SDUMP macro instruction provide several important options including: the type of entry to the dumping routine, the address spaces to be dumped, the use of the SQA buffer, if default SDATA areas should be included in the dump, and a reason code option for dumps that fail.

**BRANCH Options:** Issuers of the SDUMP macro instruction can specify branch entry (BRANCH=YES) or SVC entry (BRANCH=NO) to the dumping routine. Callers who specify an SVC entry must satisfy one of the following conditions:

- Be APF authorized
- Be in supervisor state
- Be executing in a system key

Callers who cannot issue an SVC must specify branch entry on the SDUMP macro instruction to obtain an SVC dump. The branch entry caller must be in key 0, supervisor state, and must also satisfy one of the following conditions:

- Be in SRB mode
- Hold any lock
- Be disabled (with a supervisor bit on in the PSASUPER field of the prefixed save area)
- Have an enabled-unlocked-task FRR on the FRR stack

The branch entry interface uses standard linkage conventions. On entry, register 13 must point to a 72-byte save area. Branch entry callers must include the CVT mapping macro instruction with the PREFIX=YES parameter.

**Address Space Options:** There are several options that allow a user to dump one or more address spaces up to a limit of 15. These options are obtained by specifying the following parameters of the SDUMP macro instruction:

ASID
ASIDLST
LISTA
TYPE=XMEM
TYPE=XMEME
SUBPLST

Except for SUBPLST, these parameters automatically cause an SRB to be scheduled to produce the dump. This type of dump is called a scheduled dump. If any ASID in the subpool list specified by SUBPLST is different from the current ASID, SUBPLST also produces a scheduled dump; if all of the ASIDs in the subpool list are not different from the current ASID,

SUBPLST produces a synchronous dump. A synchronous dump is a dump that is finished when control returns to the caller rather than a dump that is still in progress when control is returned to the caller.

**SQA Buffer Option:** Callers who specify the BUFFER=YES parameter of the SDUMP macro instruction will obtain a dump of a 4K buffer reserved in the system queue area (SQA) for the callers of SVC dump. A user can reserve the buffer (by setting the high order bit of the CVTSDBF field of the communication vector table (CVT)) and fill it with information before invoking SVC dump. The buffer should be used by routines that are involved with volatile data that would be changed or must be changed before SVC dump can dump it.

The CVTSDBF field of the CVT points to the buffer. Before using the buffer, callers of SVC dump must check that the high order bit of CVTSDBF is off, using compare and swap logic. If the bit is set, assume that a dump is in progress and continue processing as if a dump could not be taken. If the bit is not set, set the bit before filling the buffer and calling SVC dump.

**Default SDATA options:** When a caller must limit the amount of data in the dump and knows exactly what information is required to diagnose a problem, the caller can invoke SDATA specifying NODEFAULTS to override the normal SDATA options. These default options include:

- ALLPSA
- SQA
- SUMDUMP
- IO
- Default SDATA options specified by the CHNGDUMP command.

**Failing Dump option:** Callers who specify the TYPE=FAILRC parameter on the SDUMP macro receive special information from SVC DUMP whenever the dump fails. When control returns to the caller after a dump failure and TYPE=FAILRC was specified, the reason code is combined with the return code and passed to the caller in both register 15 and the ECB. When the return code is in the ECB, the POST flag is set on. The values and meanings of these codes follow:

| Value | Meaning |
|---|---|
| 00000208 | Another dump was in progress at the time of request. |
| 00000308 | DUMP=NO was specified at IPL time or CHNGDUMP SET,SDUMP,NODUMP was specified. |
| 00000408 | Dump was suppressed by the SLIP NODUMP facility. |
| 00000508 | No SYS1.DUMP data set was available. |
| 00000608 | I/O error occurred writing the first record. |
| 00000808 | No SRBs could be scheduled to start the dump. |
| 00000908 | A termination error occurred in SVC DUMP before the first record could be written. |
| 00000A08 | No dump could be taken because of a STATUS STOP SRBs condition, which prevents SVC DUMP from scheduling any SRBs or doing I/O. |
| 00000B08 | The dump was suppressed because the dump analysis and elimination facility determined that the problem was a duplicate of a known problem. |

### Determining the Initial Status of an SVC Dump Request

Users can determine the initial status of an SVC dump request by examining the SDWASDRC byte in the system diagnostic work area (SDWA). The possible values of this indicator and their meanings follow:

| Hexadecimal Value | Meaning |
|---|---|
| 0 | No SVC dump was requested. |
| 1 | An SVC dump was successfully started. |
| 2 | An SVC dump was suppressed because another SVC dump was in progress. |
| 3 | An SVC dump was suppressed by a request by the installation (for example: DUMP=NO at IPL or CHNGDUMP SET,SDUMP,NODUMP). |
| 4 | An SVC dump was suppressed by a SLIP NODUMP command. |
| 5 | An SVC dump was suppressed because a SYS1.DUMP data set was not available. (Only for synchronous dumps) |
| 6 | An SVC dump was suppressed because an I/O error occurred during the initialization of the SYS1.DUMP data set. (Only for synchronous dumps) |
| 8 | An SVC dump was suppressed because an SRB could not be scheduled to activate the dump tasks in the requested address spaces. |
| 9 | An SVC dump was suppressed because a terminating error occurred in SDUMP before the first dump record was written. |
| A | An SVC dump was suppressed because a status stop SRB condition was detected. (This prevents dump I/O from completing.) |
| B | An SVC dump was suppressed by DAE. |
| C-FE | Reserved. |
| FF | An SVC dump was suppressed for some other unspecified reason. |

## Obtaining an SVC Dump

The two types of SVC dumps are a scheduled dump and a synchronous dump. The type of dump produced depends on the dump options in the SDUMP parameter list and the mode of entry (SVC entry or branch entry).

### Scheduled Dump

A scheduled dump is a dump that is scheduled by an SRB. It is produced in the following cases:

- BRANCH=YES is specified on the SDUMP macro instruction.
- BRANCH=NO is specified (or used as a default) on the SDUMP macro instruction along with one or more of the following parameters:
  - ASID or ASIDLST
  - LISTA
  - TYPE=XMEM or TYPE=XMEME
  - SUBPLST (with any ASID different from the ASID of the current address space)

Callers interested only in the current address space should not use the ASID option. If the current ASID is specified as a parameter on the SDUMP macro instruction, the caller will obtain a scheduled dump and not a synchronous dump.

If a caller obtaining a scheduled dump does not specify the ECB option and wait for the dump to complete, the dump is produced asynchronously with the caller's recovery. For this reason, a scheduled dump is sometimes called an asynchronous dump. When BRANCH=YES is specified, the dump is also called a branch-entry SDUMP. The dump taken by SLIP is a branch-entry SDUMP.

A scheduled dump can contain one of three types of summary dumps. See the section "Obtaining a Summary Dump" for a description of the types of summary dumps.

### Synchronous SDUMP

A synchronous SDUMP is a non-scheduled dump that is completed before the caller regains control from SVC 51. This type of dump is always entered by an SVC (BRANCH=NO) and is sometimes called an SVC-entered SDUMP. A SYSMDUMP is an example of a synchronous dump. It is requested by an SVC 51 during ABDUMP processing.

In the case of a synchronous dump, the caller should not specify the ECB option, because there is nothing to wait for.

A synchronous dump can contain only an enabled summary dump. See the topic "Obtaining a Summary Dump" for a description of this type of dump.

## Obtaining a Summary Dump

A summary dump will be obtained by default unless the CHNGDUMP command specifies NOSUM, or SDATA=NOSUM or SDATA=NODEFAULTS is specified as a parameter on the SDUMP macro instruction. In order to avoid duplicate data in a summary dump, the dumping routine uses a range table to accumulate the addresses of the storage areas to be dumped. Duplicate storage is eliminated only around the PSW and registers. If the SDWA address is in a register, it is dumped once as the SDWA in formatted output, and once as unformatted storage.

The type of summary dump depends on two SDUMP parameters, BRANCH and SUSPEND. (The default for both of these parameters is NO.) The summary dumps and the parameters that produce them are:

| Summary Dump | Parameters |
|---|---|
| DISABLED | BRANCH=YES, SUSPEND=NO |
| SUSPEND | BRANCH=YES, SUSPEND=YES |
| ENABLED | BRANCH=NO (SUSPEND cannot be specified) |

A description of each of the three types of summary dumps follows.

**Disabled Summary Dump:** The purpose of the disabled summary dump is to save volatile system information before returning control to the user. The caller can specify either the SUMLIST or SUMLSTA parameter and the PSWREGS parameter to save specific information in the summary dump. However, because the system is disabled, the dump includes only data that is paged in.

A disabled summary dump contains the following information:

- The cross memory status record giving the home, primary, and secondary ASIDs and the CML ASID (if the caller holds a CML lock)
- The storage areas specified by the parameters SUMLIST and SUMLSTA

- 4K of storage before and 4K after the PSW address from primary storage, and 4K of storage before and 4K after each of the general purpose registers from primary and secondary storage, as provided by the caller's PSWREGS area. If the control registers are provided, they will be used to determine primary and secondary storage. Otherwise, the storage will be dumped from the caller's primary and secondary address space.
- The PSA, PCCA, and LCCA for each processor
- The current PCLINK stack (pointed to by PSASEL)
- The IHSA and its associated XSB and PCLINK stack (The PSW and registers from the IHSA are added to the range table causing 4K of storage around each address to be dumped.)
- If it exists, the caller's SDWA (The PSW and register addresses from the SDWA are added to the range table causing 4K of storage around each address to be dumped.)
- The SUPER FRR stacks
- The global, local, and CPU work save area (WSA) vector tables and the save areas pointed to by entries in each WSA vector table
- 4K of storage on either side of the address portion of the I/O old PSW, the program check old PSW, the external old PSW, and the restart old PSW

**Suspend Summary Dump:** The purpose of the suspend summary dump is to save volatile system information before returning to the caller. The caller can specify either the SUMLIST or SUMLSTA parameter and the PSWREGS parameter to save specific information in the summary dump. The difference between the suspend summary dump and the disabled summary dump is that the suspend summary dump can save pageable data.

A suspend summary dump contains the following information:

- The ASID record, PSA, PCCA, LCCA, IHSA, XSB, and PCLINK stack
- The storage area specified by the parameters SUMLIST and SUMLSTA
- 4K of storage before and 4K after the PSW address from primary storage, and 4K of storage before and 4K after each of the general purpose registers from primary and secondary storage, as provided by the caller's PSWREGS area. If the control registers are provided, they will be used to determine primary and secondary storage. Otherwise, the storage will be dumped from the caller's primary and secondary address space.
- The caller's ASCB
- The caller's unit of work (a TCB, RB, and XSB or SSRB, XSB, and PCLINK stack)
- For TCB mode callers, the caller's SDWA, RTM2 work areas, the associated SDWAs, the PSW, and registers
- For SRB mode callers, the SDWA (The PSW and register addresses from the SDWA)
- The caller's register save area is added to the range table causing 4K of storage around each register address to be dumped.

**Enabled Summary Dump:** The purpose of the enabled summary dump is to group information for debugging dumps by specifying a particular option on the SDUMP macro instruction. This is the only type of summary dump that can be produced by an SVC entry to SDUMP. If the dump is a scheduled dump, the summary information is saved for each address space specified.

An enabled summary dump contains the following information for each address space:

- The ASID record (contains ASID, jobname, and stepname)
- The storage areas specified by the parameters SUMLIST and SUMLSTA
- 4K of storage before and 4K after the PSW address from primary storage, and 4K of storage before and 4K after each of the general purpose registers from primary and secondary storage, as provided by the caller's PSWREGS area. If the control registers are provided, they will be used to determine primary and secondary storage. Otherwise, the storage will be dumped from the caller's primary and secondary address space.
- The RTM2 work areas pointed to by all TCBs in the address space
- 4K of storage before and 4K after the PSW in each RTM2WA at the time of the error
- 4K of storage before and 4K after each register in each RTM2WA at the time of the error

## Suppressing SDUMPs and SYSMDUMPs

You can use dump analysis and elimination (DAE) to suppress SDUMPs and SYSMDUMPs. Use of DAE reduces the number of duplicate dumps and avoids tracking and screening of unnecessary dumps. System performance is improved because duplicate problems are recognized before the dump is taken. Another benefit is that the symptom data, stored in the SYS1.DAE data set, provides a consistent set of data for identifying a failure.

When you use DAE, it does the following for each dump taken:

- Builds a symptom string from error information contained in either of the following:
  - A user-supplied symptom record (as defined by the ADSR mapping macro) that contains unique information about the dump (DAE checks for this first). To allow DAE to use such a symptom record, invoke the SDUMP macro with the SYMREC parameter.
  - A system diagnostic work area (SDWA). DAE checks for an SDWA only if there is no user-supplied symptom record. An SDWA is available in a recovery environment (that is, the system passes an SDWA to an ESTAE- or FRR-type recovery routine).
- Stores each symptom string it creates in SYS1.DAE.
- Compares each symptom string it creates to those previously recorded in SYS1.DAE to determine if the dump should be suppressed.
- Suppresses the dump under the following circumstances:

  **User-supplied symptom record:**

  - The symptoms of the dump must match existing symptoms recorded in SYS1.DAE.

  **SDWA:**

  - The symptoms of the dump must match existing symptoms recorded in SYS1.DAE, **and**
  - The VRADAE indicator in the SDWAVRA must be set. This condition is met only if your program supplies a recovery routine that invokes the VRADATA macro with KEY=VRADAE. Specification of this key indicates to DAE that the SDWA contains sufficient data to uniquely identify the problem.

**Notes:**

1. DAE will build a symptom string only if certain minimum requirements are met (see *SPL: System Modifications* for a description of these requirements).
2. If DAE does not find either a user-supplied symptom record or an SDWA, it will not suppress the dump.

The following publications contain additional information about DAE:

- *Operations: System Commands* contains the syntax and use of the SET DAE command.
- *SPL: System Modifications* provides information about how an installation can modify DAE to fit its needs.
- *Debugging Handbook* contains sample symptom output, DAE control block information, and field descriptions provided by the ADSR mapping macro.
- *System Logic Library* provides a description of the logic and an explanation of symptom strings.
- *SPL: Initialization and Tuning* contains information about the ADYSETxx parmlib members that are used by DAE.

## Using Dump Data Sets

The SDUMP macro instruction, the DUMP command, and the ACTION=SVCD or TRDUMP parameter of the SLIP command produce SVC dumps that are stored in dump data sets. A dump data set can be placed on a direct access storage device (allocated with RECFM=F, LRECL=4104) or on a tape. The system obtains these data sets by using the DUMP parameter of the system parameter list (at IPL) or by using the DUMPDS operator command (after IPL). (For details on the DUMP parameter of the system parameter list, see *Initialization and Tuning*. For a description of the DUMPDS operator command, see *Operations: System Commands*.)

Dump data sets are dynamically allocated to the DUMPSRV address space with DISP=SHR. Users must do one of the following:

- If you are running another job using these data sets (for example, AMDPRDMP service aid), specify DISP=SHR when allocating the dump data set (This will share the dump data set with the DUMPSRV address space.)
- Issue the DUMPDS operator command to remove the data set from use by the system

## Using the Dumping Services Commands

There are several operator commands related to controlling, taking, displaying, and suppressing dumps. For complete details and the syntax of each of these commands, see *Operations: System Commands*.

A brief description of the use of these commands follows:

| Command | Use |
|---|---|
| CHNGDUMP | Modifies or overrides the default dump options used on SYSABEND, SYSUDUMP, SYSMDUMP, and SDUMP. |
| DISPLAY DUMP | Displays the current dump options set by the CHNGDUMP command and some specific information from the dump header record. It also indicates whether the dump data sets are full or empty and includes a timestamp. |
| DUMP | Obtains a scheduled dump for a job or address space. It can also obtain a dump of the system for performance analysis. |
| DUMPDS | Modifies the SDUMP data set queue and thereby adds or deletes data sets from use by SDUMP. It also clears full dump data sets. |
| SET DAE=xx | Depending on the keywords specified (by the SYS1.PARMLIB member ADYSETxx), starts or stops DAE processing or requests that SDUMPs or SYSMDUMPs be suppressed, matched, and/or updated. |

## Canceling and Restarting the DUMPSRV Address Space

If an installation is using post dump exits, cancelling the DUMPSRV address space can be useful. The cancellation, followed by the resource manager's restart causes new versions of the post dump exits to be loaded and used. See *SPL: System Modifications* for additional information concerning an installation's use of post dump exits.

## Getting More Than One SYSMDUMP

MVS/XA can handle a SYSMDUMP in two ways. One makes only the most recent SYSMDUMP available; each dump after the first overlays the preceding one. The other makes only the first SYSMDUMP available; any subsequent dumps are lost. If you choose this second method, however, you can take additional steps to avoid losing subsequent SYSMDUMPs.

To make only the first SYSMDUMP available, you must specify DSNAME=SYS1.SYSMDPxx,DISP=SHR on the SYSMDUMP DD statement. (See *Job Control Language* for more information about the SYSMDUMP DD statement.) Any other value for DSNAME or DISP causes the system to make only the most recent SYSMDUMP available.

When the data set is created, it must be cleared and an EOF must be written as the first record on the SYSMDPxx data set. EOF means that the data set is empty; anything other than EOF means that the data set is full and subsequent dumps are lost. To obtain these subsequent SYSMDUMPs, you must intercept the message

```
IEA993I SYSMDUMP TAKEN TO data set name
```

and pass control to an installation-written routine that will copy the dump onto another data set, clear the data set, and write another EOF as the first record on the SYSMDPxx data set. One way to intercept message IEA993I is to use the WTO exit routine IEECVXIT. (See *SPL: User Exits* for information on message routing exit routines. *SPL: System Modifications* contains information on post dump exit processing, which can be used to off-load SYSMDUMP data sets.)

## Providing Recovery Routines

When a unit of work (either a task or an SRB) terminates abnormally, RTM invokes a recovery routine, if one exists. The recovery routine might determine where the error occurred and decide whether to continue with termination or to continue processing (retry) at some appropriate point in the main routine. Other functions that your recovery routine might perform are to document the error, take a dump of the storage necessary to determine the cause of the error, and, when required, free resources that are no longer needed. See "Providing Information for DAE" later in this section for information about documenting the error.

The decision about whether or not your particular function requires recovery depends, naturally enough, on the nature of the function. One major factor in this decision is the set of resources that the function acquires. If a function acquires resources that might be requested by another function or that are not known to be related to the task or SRB then a recovery routine should be established to free the resources. Examples of resources that must be freed are queues of control blocks serialized for exclusive use, private locks, cells obtained from cell pools, and storage that must be explicitly freed.

Once you have determined that your particular function requires recovery, you must select the type of recovery routine that your function needs -- either a functional recovery routine (FRR) or an ESTAE-type recovery routine.

Because of the complexity of the MVS/XA environment and the various uses your installation might make of that environment, it is difficult to provide many hard and fast rules about which type of recovery routine is best for a particular situation. If the recovery routine requires that a lock held by the main routine not be freed, or if cross memory mode is a factor, the recovery routine must be an FRR. In other situations generally grouped as task recovery, an ESTAE-type recovery routine can provide the function you need.

In general, establish an FRR to provide recovery for locked, disabled, or SRB mode routines or routines executing in cross memory mode. Identify an FRR to RTM by coding the SETFRR macro instruction. This macro instruction creates an entry in a system recovery area called an FRR stack. When RTM invokes an FRR, the FRR runs with the locks that were held at the time of the error, or as modified by previous FRRs, and the enablement implied by those locks.

In general, establish an ESTAE-type recovery routine to provide recovery for unlocked tasks that do not execute in cross memory mode. Identify an ESTAE-type recovery routine to RTM by coding the ESTAE or FESTAE macro instruction or the ESTAI parameter on the ATTACH macro instruction.

**Note:** The STAE macro instruction and the STAI parameter of the ATTACH macro instruction are available for compatibility with Release 1 of VS2 and with MFT and MVT. "STAE/STAI Exit Routines" later in this chapter provides information on STAE and STAI; however, it is recommended that you use ESTAE, FESTAE, or ESTAI for all new development.

"Selecting a Recovery Routine" later in this chapter describes the factors, particularly the system environment, that you must consider in choosing the type of recovery routine for a particular situation. Which type of recovery routine you choose also affects the order in which your recovery routine can get control. When percolation occurs, recovery routines get control in LIFO (last-in, first-out) order, starting with FRRs (if any). If all FRRs percolate, then all ESTAE-type recovery routines get control in the reverse of the order in which they were established. See "Percolation, Retry, and Resume" later in this chapter for detailed information on the process of percolation, the factors that go into making a request for a retry to the main routine, and a definition of the special situation when an FRR can request a resume.

In some situations, the function that your recovery routine must perform is itself one that requires recovery. See "Recovery for Recovery" later in this chapter for a description of such situations and some of the considerations involved in providing recovery for a recovery routine.

"Recovery Routine Guidelines" pulls together various pieces of information for you to use as a basis for making decisions about recovery routines and designing recovery routines.

## Providing Information for Dump Analysis and Elimination

Dump analysis and elimination (DAE) depends on information that users provide in ESTAE and FRR recovery routines to construct unique symptom strings needed to describe software failures. DAE uses these symptom strings to analyze dumps and suppress duplicates as requested. The symptom string contains symptoms (specific pieces of information) that DAE obtains from the system diagnostic work area (SDWA), SDWA extensions, ABDUMP symptom area, and the SDWA variable recording area (SDWAVRA).

If you are placing information into the SDWAVRA for use by DAE, you must provide the information in key/length/data format. You can use the VRADATA macro to create the entry in this format.

Users must select symptoms carefully. If the data they supply is too precise, no other failure will have the same symptoms; if the data is too general, many failures will have the same symptoms. See "Suppressing SDUMPs and SYSMDUMPs" earlier in this section for additional information about DAE and for a list of other publications that document DAE.

## Selecting a Recovery Routine

Several basic restrictions and requirements govern your choice between an FRR and an ESTAE-type recovery routine. These are:

- An FRR can be established only when you make the request in key 0 and supervisor state.
- If an enabled unlocked task establishes an FRR using EUT=YES, it cannot issue any SVCs except SVC 13. Also, the system does not dispatch any new asynchronous exits on that task until all FRRs for the task have been deleted.
- From the time an FRR is established until the time it is deleted, at least one of the following must be true.
  - Some lock is held.
  - The unit of work is executing either disabled or as an SRB.
  - An FRR with EUT=YES exists.
- The size of each FRR stack satisfies the recovery needs of the control program. If additional FRRs placed on the stack cause the size to be exceeded, the routine issuing the SETFRR macro instruction terminates abnormally. Any user-written routines outside of the control program may add one, and only one, FRR to the stack; if they add more than one, abnormal termination occurs if the size of the stack is exceeded. This applies to all of the recovery stacks, including the normal stack. The normal FRR stack is used by control program routines that are invoked on behalf of the user.
- You cannot use the ESTAE macro instruction or the ESTAI parameter of the ATTACH macro instruction to establish a recovery routine when executing in cross memory mode or as an SRB.
- You can use the FESTAE macro instruction in cross memory mode as long as data addressability at the time the macro is issued is to the dispatched address space (home).

Beyond these basic restrictions and requirements, your choice of an FRR or an ESTAE-type recovery routine depends on the environment for which recovery is to be established and on the environment in which the recovery routine is to get control. Other major differences are the register interface and the availability of a system diagnostic work area (SDWA) for use by the recovery routine.

The following information under "System Environment," "Register Interface," and "System Diagnostic Work Area (SDWA)" describes the differences between FRRs and ESTAE-type recovery routines in more detail. And, if you choose an ESTAE-type recovery routine, you must decide whether to establish it with the ESTAE macro instruction, the FESTAE macro instruction, or the ESTAI parameter of the ATTACH Macro instruction. See "ESTAE-type Recovery Routines" for the basic differences between the three methods of establishing an ESTAE-type recovery routine.

## System Environment

The environment -- the state of the system -- consists of such factors as locks held, enablement or disablement, supervisor or problem program state, PSW key, PSW key mask (PKM), authorization index, and cross memory mode. To enable you to determine whether an FRR or an ESTAE-type recovery routine best meets your needs, identify the system environment your recovery routine requires and use the following information to select the type of recovery routine.

### LOCKING

**ESTAE-Type Recovery Routines:** All locks are freed before the first ESTAE-type recovery routine gets control.

**FRRs:** The locks held when the first FRR gets control are the same as they were at the time of the error.

If the recovery routine is to free locks held by the main routine, the FRR should issue the SETRP macro instruction to use the lock-freeing functions of RTM.

The lock-freeing functions of RTM are effective only for percolation, not for retry. In any event, the FRR must not free the last global lock and it must not free any local lock because RTM depends on these locks to serialize the system diagnostic work area (SDWA) passed to the FRR. Freeing these locks can cause different units of work to use the same SDWA simultaneously.

For more information on how RTM manipulates locks, see "Decisions Made in a Recovery Routine" later in this chapter.

### DISABLEMENT

**ESTAE-Type Recovery Routines:** All ESTAE-type recovery routines are entered enabled.

**FRRs:** An FRR gets control and is entered disabled if, at the time of the error, the main routine is disabled and any of the following states exist:

1. Any global spin lock is held.
2. A super FRR stack was active or the control program set a bit in the PSASUPER area of the PSA. (Generally, this condition occurs when an interrupt handler is running.)

### ADDRESSING MODE

The addressing mode for all recovery routines (both ESTAE and FRR) is the addressing mode of the caller at the time the routine was established.

### SUPERVISOR/PROBLEM PROGRAM STATE

**ESTAE-Type Recovery Routines:** An ESTAE-type recovery routine is entered in the state -- either supervisor or problem program -- that existed at the time it was established.

**FRRs:** All FRRs are entered in supervisor state.

### AUTHORIZATION INDEX (AX)

The authorization index for all recovery routines -- both ESTAE-type recovery routines and FRRs -- is the effective AX for the address space in which the recovery routine is to get control. Refer to "Cross Memory Authorization" in the Inter-Address Space Communication section for more information about the AX.

### PSW KEY

**ESTAE-Type Recovery Routines:** An ESTAE-type recovery routine is entered with key 0 whenever it was established with a key less than 8. Otherwise, it is entered with the key that existed at the time it was established.

**FRRs:** All FRRs are entered in key 0.

### PSW KEY MASK (PKM)

**ESTAE-Type Recovery Routines:** All ESTAE-type recovery routines are entered with a PSW key mask (PKM) that is the ORing of the following:

- The PSW key under which the exit is to get control.
- The TCBPKF.
- The PKM that existed when the ESTAE-type recovery routine was established.

However, if the recovery routine was established with a FESTAE macro instruction, the system uses the PKM that existed at the time of the error.

**FRRs:** See the following information on "Cross Memory State" for information about the PKM that exists when an FRR is entered.

### CROSS MEMORY STATE

**ESTAE-Type Recovery Routines:** All ESTAE-type recovery routines are entered in primary mode with the primary address space (PASID) and the secondary address space (SASID) the same as the dispatched address space (HASID).

**FRRs:** The cross memory environment for an FRR depends on the values coded for the MODE parameter in the SETFRR macro instruction that established the FRR.

### NORMAL Addressing Environments

Specifying HOME, PRIMARY, or FULLXM for the MODE parameter of the SETFRR macro instruction indicates to RTM 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 in home mode; that is, the FRR is entered with PASID=SASID=HASID, in primary mode. The PSW key mask (PKM) for an FRR that covers SRB code is the same as the PKM at the time of the error. The PKM for an FRR that covers a task is the TCBPKF transformed into a PKM.

**MODE = PRIMARY**

If you code MODE = PRIMARY, the FRR gets control in primary mode with the primary and secondary address space the same as the primary address space that existed when the SETFRR macro instruction was issued. The PKM is the PKM that existed when the SETFRR macro instruction was issued.

**MODE = FULLXM**

If you specify MODE = FULLXM, the FRR gets control in the cross memory environment that existed when the SETFRR macro instruction was issued. That is, the primary address space (PASID), secondary address space (SASID), the mode, and PKM are the same as those that existed when the SETFRR macro instruction was issued.

If RTM cannot enter an FRR with its normal addressing environment established as defined by the MODE parameter on the SETFRR macro instruction, RTM bypasses the FRR and percolates to the next FRR on the FRR stack, unless the FRR was established to execute in a restricted addressing environment (also specified by the MODE parameter).

## RESTRICTED ADDRESSING ENVIRONMENTS

Specifying either LOCAL or GLOBAL (or both) for the MODE parameter of SETFRR indicates to RTM that the FRR can run in a restricted addressing environment. When an FRR is entered in a restricted addressing environment, it is often called a resource manager because its only purpose to recover resources. These resources can be critical system resources (GLOBAL) or critical address space resources (LOCAL). RTM does not allow an FRR to retry.

When RTM enters an FRR to recover critical resources, it sets bits in the SDWA to indicate to the FRR in which of the restricted addressing environments it is being entered.

If you specify both LOCAL and GLOBAL, RTM first tries to enter the FRR in the LOCAL restricted addressing environment. If it cannot, RTM then tries to enter the FRR in the GLOBAL restricted addressing environment. These environments, and the bits that RTM sets to indicate the environment to the routine, are:

**MODE = GLOBAL:** MODE = GLOBAL should be used by services that need to clean up global resources if the main routine terminates abnormally. When GLOBAL is specified and RTM cannot enter the FRR in its normal mode, it enters the FRR in the restricted GLOBAL mode if one of the following system conditions exists:

- The main routine holds a global spin lock.
- A "super FRR stack" was active or the control program has set a bit in the PSASUPER word of the PSA. (Generally this condition occurs when an interrupt handler is running.)

A global FRR must reside in commonly addressable storage.

When RTM enters an FRR in restricted GLOBAL mode, the entry environment is:

- PASID = SASID, in primary mode. The PASID can be that of **any** address space; thus, when entered in this mode, the routine must not reference private storage.
- RTM sets the SDWAGLBL bit in the SDWA to one to indicate that RTM is entering the routine in restricted GLOBAL mode (otherwise, this bit is set to zero).
- RTM sets the SDWACLUP bit in the SDWA to one to indicate that the routine is not allowed to retry, although, if system conditions permit, subsequent FRRs are permitted to retry.

**MODE = LOCAL:** 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 RTM 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.

RTM enters an FRR established with the LOCAL parameter in LOCAL restricted mode for two different reasons:

1. RTM tried to establish the environment required to enter the FRR in normal mode but could not; this problem can occur, for example, when the SASID 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 PASID and SASID are the same as the locked address space.
- RTM sets the SDWALCL bit in the SDWA to one to indicate that RTM is entering the routine in LOCAL restricted mode (otherwise, this bit is set to zero).
- RTM sets the SDWALCL and SDWAGLBL bits if the SETFRR macro instruction included both the LOCAL and GLOBAL parameters, and RTM is entering the FRR in LOCAL restricted mode while the LOCAL lock is held and one of the following is true:
  - A spin lock is held
  - A super bit is set
  - The FRR is on a super FRR stack
- RTM sets the SDWACLUP bit in the SDWA to one to indicate that the routine is not allowed to retry, although, if system conditions permit, subsequent FRRs are permitted to retry.

The following considerations apply when RTM enters the FRR in LOCAL restricted mode as a result of the second reason stated earlier.

1. If the FRR issues a SETRP macro instruction to request that RTM 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.
2. The FRR must not depend on executing in task mode because, even though the SETFRR macro instruction was issued in task mode, an FRR entered in LOCAL restricted mode executes in SRB mode. In this case, the information in the SDWA reflects the interrupted process (from IHSA or SSRB) that originally held the CML lock, and the FRR is not permitted to retry.
3. 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 executing FRR has established 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.

## REGISTER INTERFACE

**ESTAE-Type Recovery Routines:** Before entering an ESTAE-type recovery routine, RTM attempts to obtain and initialize a system diagnostic work area (SDWA). The SDWA contains information about the error; see "System Diagnostic Work Area (SDWA)" later in this section. The first word of the SDWA contains the address of the parameter list specified on the ESTAE-type recovery request. The register interface to the recovery routine varies depending on whether or not RTM can obtain an SDWA.

If RTM can obtain an SDWA, the register contents on entry to the recovery routine are:

Register 0 a code indicating the type of I/O processing performed:
- 0 active I/O has been quiesced and is restorable.
- 4 active I/O has been halted and is not restorable
- 8 no active I/O at ABEND time
- 16 no I/O processing was performed

Register 1 address of the SDWA
Register 13 save area address (72 bytes)
Register 14 return address
Register 15 entry point address of the ESTAE recovery routine

The contents of all other registers are unpredictable.

If RTM cannot obtain an SDWA, the register contents on entry to the recovery routine are:

Register 0 a decimal 12 to indicate that an SDWA was not obtained.
Register 1 ABEND completion code
Register 2 address of user-supplied parameter list
Register 13 unpredictable
Register 14 return address
Register 15 entry point address of the ESTAE recovery routine

The contents of all other registers are unpredictable.

When a recovery routine entered without an SDWA completes, it must set a return code for RTM in register 15. This return code must be one of the following:

| Hexadecimal Code | Meaning |
|---|---|
| 0 | RTM is to continue with termination (percolate). |
| 4 | RTM is to schedule a retry. The recovery routine has placed the address of the retry routine in register 0. |
| 16 | Valid only for an ESTAI recovery routine. RTM is to continue with termination, and no further ESTAI processing is to be performed. |

**FRRs:** All FRRs are entered with an SDWA. The register contents on entry to the FRR are:

Register 0 address of a 200-byte work area for the FRR
Register 1 address of the SDWA
Register 14 return address
Register 15 address of the FRR

The contents of all other registers are unpredictable.

The FRR can use any register without saving its contents; however, the routine must maintain the return address supplied in register 14.

## SYSTEM DIAGNOSTIC WORK AREA (SDWA)

The system diagnostic work area (SDWA) is a communication area between RTM and the recovery routine. RTM always supplies an SDWA for an FRR. If for any reason RTM cannot supply an SDWA, it bypasses the FRR and percolates to the next FRR on the FRR stack. Because RTM cannot always supply an SDWA for an ESTAE-type recovery routine, an ESTAE-type recovery routine must always check the contents of register 0 to determine if there is an SDWA available.

The mapping macro IHASDWA provides the field names and describes their content and use. Detailed information on the name, offset, and meaning of each field appears in the *Debugging Handbook*. To enable you to determine what kinds of information RTM supplies to the recovery routine in the SDWA and what kinds of information the recovery routine must return to RTM, Figure 47 lists the names and meanings of some of the key fields.

It is recommended that you use the SETRP macro instruction, which manipulates fields in the SDWA, to change fields in the SDWA before returning control from your recovery routine to RTM.

| Field Name | Use |
|---|---|
| SDWAPARM | This 4-byte field, located at offset 0, contains the pointer to the user parameter list supplied by the user for an ESTAE-type recovery routine. For an FRR, this field contains the address of the six-word parameter area returned by a SETFRR macro instruction with the PARMAD parameter. |
| SDWACMPC | This field contains the ABEND completion code that existed when RTM entered the recovery routine. The recovery routine can change the ABEND code by altering this field. The system code appears in the first twelve bits, and the user code appears in the second twelve bits. |
| SDWACRC | This field contains the error code associated with the ABEND completion code in SDWACMPC. The recovery routine can change the reason code by altering this field by means of the SETRP macro instruction and its REASON keyword. This field is not defined if the SDWARCF flag field is zero. |
| SDWAGRSV | This field shows the contents of the general purpose registers (0-15) as they were at the time of the error. |
| SDWAEC1 | This field contains the extended control (EC) PSW that existed at the time of the error. |
| SDWAEC2 | The contents of this field vary according to the type of recovery routine:<br>• For an ESTAE/FESTAE routine, the field contains the PSW of the RB that created the recovery routine at the time the RB last incurred an interruption. However, if this RB was disabled or was holding a lock when the interrupt occurred, the field does not contain the last interrupt address, but the address of the second to last interrupt.<br>• For an ESTAI routine, this field contains zeros.<br>• For an FRR, the field contains the extended control PSW used to give control to the FRR. |

Figure 47 (Part 1 of 2). Key Fields in the SDWA

| Field Name | Use |
|---|---|
| SDWASRSV | The contents of this field vary according to the type of recovery routine:<br>• For an ESTAE/FESTAE routine, this field contains the general purpose registers (0-15) of the RB that established the recovery routine as they were at the time the RB last incurred an interrupt.<br>• For an ESTAI routine, this field contains zeros.<br>• For an FRR, this field has the same contents as SDWAGRSV.<br>If the recovery routine requests a retry, RTM uses the contents of this field to load the registers for the retry routine. To update the contents of the registers for the retry routine, you must make the required changes to SDWASRSV and request a register update using the RETREGS=YES parameter on the SETRP macro instruction. You can update the registers directly or with the RUB parameter on SETRP. |
| SDWASPID | This field contains the subpool ID of the storage used to obtain the SDWA. |
| SDWALNTH | This field contains the length, in bytes, of this SDWA, the SDWA extensions, and the variable recording area. (This allows the user to free the extensions along with the SDWA.) |
| SDWACOMU | The recovery routines can use this eight-byte field to communicate with each other when percolation occurs. RTM copies this field from one SDWA to the next on all percolations. When the field contains all zeros, either no information is passed or RTM has not been able to pass the information. |
| SDWAFAIN | This 12-byte field contains the six bytes of the instruction stream that both precede and follow the failing instruction pointed to by the PSW. The SDWAFAIN field contains zeroes if RTM cannot access the failing instruction stream pointed to by the time-of-error PSW. For example, if the time-of-error PSW is not valid, the SDWAFAIN field contains zeroes. |
| SDWADAET | This eight-byte field contains DAE status and error flags for this dump. |
| SDWAOCUR | This two-byte field contains the current count of the number of previous occurrences of these symptoms in other SDWAs. |
| SDWAVRAL | This field contains the length of the variable recording area (VRA) for this SDWA. |
| SDWAHEX | This bit is set by the recovery routine to indicate that EREP is to print the data in the VRA in hexadecimal form. |
| SDWAEBC | This bit is set by the recovery routine to indicate that EREP is to print the data in the VRA in EBCDIC. |
| SDWAURAL | This one-byte field is used by the recovery routine to indicate the length of the VRA used. The field is always zeroes initially, and it must be set whenever a recovery routine uses any part of the VRA. |

Figure 47 (Part 2 of 2). Key Fields in the SDWA

## ESTAE-Type Recovery Routines

You establish an ESTAE-type recovery routine (also known as an ESTAE exit or an ESTAE environment) by means of the ESTAE macro instruction, the FESTAE macro instruction, or the ESTAI parameter of the ATTACH macro instruction. Any ESTAE-type recovery routine, regardless of how it is established, executes under its own program request block created by the SYNCH service routine. The recovery routine executes in the same addressing mode as the issuer of the ESTAE macro instruction.

Before an ESTAE-type recovery routine gets control, the control program performs any purge and asynchronous processing that was specified as a macro option when the routine was established. The control program performs the requested I/O processing only for the first recovery routine. Subsequent routines receive an indication of the I/O processing previously performed, but no additional I/O processing is done. However, the control program performs asynchronous processing for each routine.

An ESTAE macro instruction creates a task-related recovery routine. A fast ESTAE (FESTAE) macro instruction performs the same function with minimal processor overhead. A recovery routine established by the FESTAE macro instruction gets control in the same sequence and under the same conditions as a recovery routine established by the ESTAE macro instruction. However, only an SVC routine executing under an SVRB (type 2,3 or 4 SVC) can issue FESTAE; other factors to be considered for using FESTAE are described later under "Using the FESTAE Macro Instruction." The ESTAI parameter on the ATTACH macro instruction establishes a recovery routine that gets control if the attached subtask encounters an unrecovered abnormal termination; that is, a recovery routine established by the ESTAI parameter gets control under the subtask. Also, any recovery routine established by the ESTAI parameter is propagated to any subsequent subtasks. Figure 48 shows the queuing structure of the ESTAE routines and the propagation of routines created by ESTAI to a subtask.

Because ESTAE-type recovery routines are associated with RBs, they are removed when their RBs terminate. The fact that they are removed is important because a program expects one of its own ESTAE-type recovery routines to get control rather than one left behind by a subprogram. A program might, however, invoke a service routine that does not create an RB. If that routine then issues an ESTAE macro instruction and fails to delete the resulting ESTAE-type recovery routine, a problem could develop if the original program is scheduled for abnormal termination. 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 macro instruction gets control.

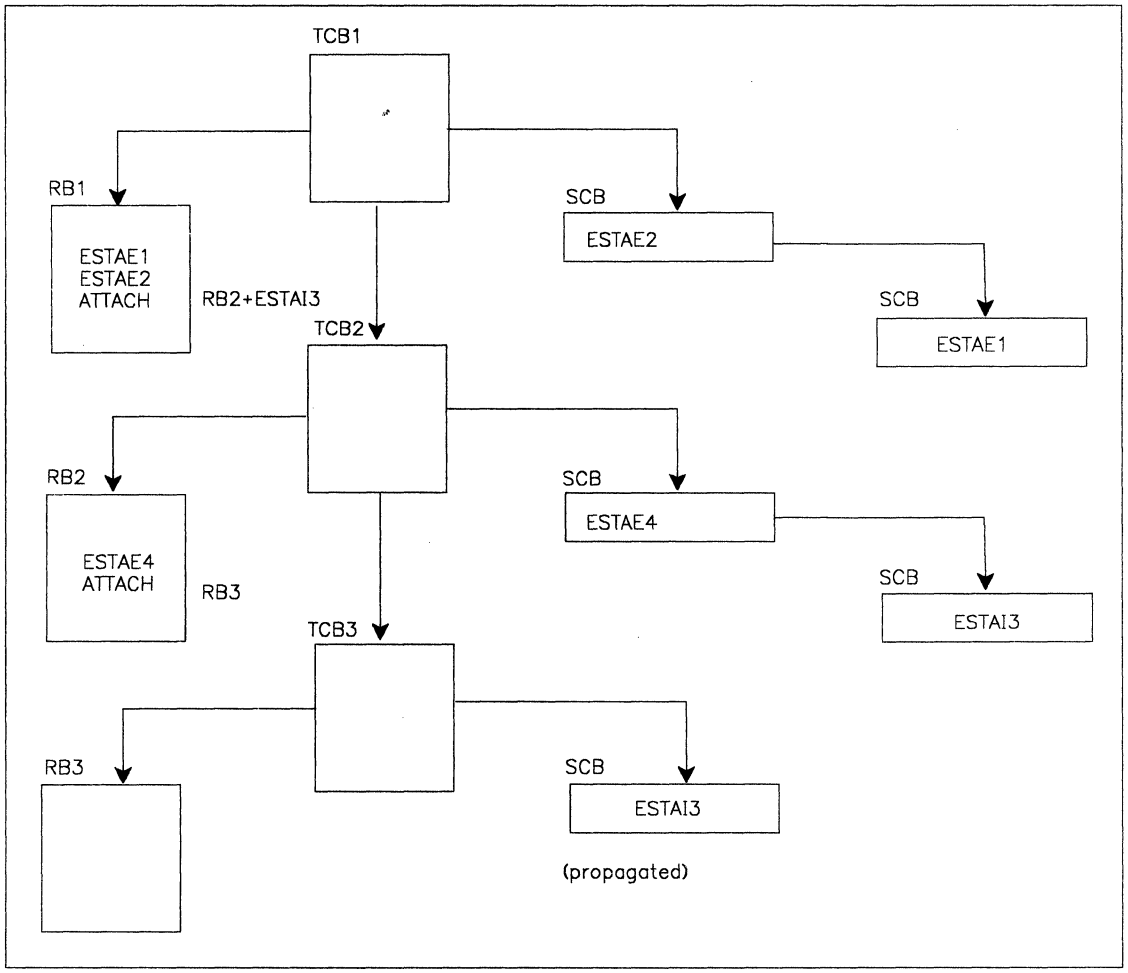



Figure 48. ESTAE Environment

This potential problem can be avoided by using the TOKEN parameter on the ESTAE macro instruction to associate a token with that ESTAE routine. In order to delete or overlay an ESTAE routine that was created with TOKEN, the same token must be specified in the ESTAE cancel or overlay macro instruction. All more recent ESTAE exits are deleted.

If a program issues an ESTAE macro instruction that specifies both the TOKEN parameter and XCTL=YES and then issues XCTL, the token must be passed as part of the parameters to the called routine so that the routine can delete the ESTAE routine.

## Using the FESTAE Macro Instruction

The FESTAE macro instruction enables a type 2, 3, or 4 SVC (an SVC for which the SVC FLIH creates an SVRB) to establish an ESTAE-type recovery routine with minimal processor overhead. However, the following restrictions apply:

- Only a type 2, 3, or 4 SVC in key 0 can use FESTAE.
- The SVC can use FESTAE only once to create a recovery routine. Therefore, any SVC needing to change its exit address must use branch entry ESTAE services, and any SVC needing more than a single recovery routine must use SVC 60 or branch entry to get the additional recovery routines.
- FESTAE can be issued in cross memory mode if the home address space is addressable at the time the FESTAE recovery routine is entered. In addition to the parameter area that you can supply by coding the PARAM parameter on the FESTAE macro instruction, a 24-byte parameter area is also available as an option. The name of the optional parameter area is RBFEPARM, and it is in the SVRB. The recovery routine receives this parameter area when an error occurs. Hence, the main 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.

FESTAE users must also include the following DSECTs for the FESTAE macro expansion:

```
IHARB
IKJTCB
IHAPSA
IHASCB
```

## Special Considerations

When writing an ESTAE-type recovery routine, consider the following:

- When an ESTAE-type recovery routine receives control, it should first examine the code in register 0 to see if an SDWA was provided. If an SDWA was not provided (that is, register 0 contains a decimal 12), register 13 does not point to a save area, and your routine must not save the registers.
- An ESTAE-type recovery routine can request, via a SETRP macro instruction parameter, that the control program free the SDWA instead of freeing it in a retry routine. When the retry routine is to free the SDWA, note that an ESTAE-type recovery routine created under any control program protection key (key 0-7) receives an SDWA in key 0 storage. Therefore, if the retry routine is executing under a key other than key 0, it must issue the MODESET macro instruction to become key 0 before issuing the FREEMAIN macro instruction to free the SDWA.
- If an ESTAE-type recovery routine itself requests termination or fails, RTM percolates and does the following:
  - 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 indicate termination, the task is terminated.

- If a non-jobstep task issues an ABEND macro instruction with the STEP parameter, RTM enters recovery for the non-jobstep task. If the recovery routines do not request a retry, the jobstep is terminated with the specified ABEND code. RTM enters subsequent recovery routines for the jobstep task only when the macro instruction that established the recovery routine specified the TERM = YES parameter.
- For some situations, RTM enters ESTAE-type recovery routines only when the TERM = YES parameter was specified when the ESTAE macro instruction was issued. The situations are:
  - System initiated logoff
  - Job step timer expiration
  - Wait time limit for job step exceeded
  - ABEND occurred because a DETACH macro instruction was issued for an incomplete subtask
  - Operator cancel
  - Error occurred on a task higher in the tree

When RTM enters the recovery routines established with the TERM = YES parameter as a result of the above errors, RTM takes the following actions:

- 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 specified a return code of 0
- Ignores any request for retry
- Ignores the TERM = YES parameter if it is specified on a nested ESTAE macro instruction

### Decisions Made in a Recovery Routine

When a recovery routine gets control, it determines why it has been entered and decides either to percolate (continue with termination), to retry, or in a special case, to resume. To convey its decision to RTM, the recovery routine issues the SETRP macro instruction, which manipulates appropriate fields in the SDWA. When the recovery routine returns to RTM, RTM honors the request, if appropriate.

The MVS/XA recovery scheme provides a parameter area for communication between the main routine and its recovery routine. The parameter area varies according to the type of recovery routine:

1. For an FRR, RTM supplies a six-word parameter area.
2. For a recovery routine established by an ESTAE or FESTAE macro instruction the user can supply a parameter area by coding the PARAM parameter on the macro instruction.

When a recovery routine is established, RTM saves a pointer to the parameter area and makes the pointer available to the recovery routine when it is entered. Usually, the main routine uses the parameter area to leave a footprint, that is, to set indicators that let the recovery routine know where in the main process the failure occurred. The recovery routine can examine the footprint to determine what action to take.

**Note:** For a recovery routine established by a FESTAE macro instruction, there is also a six-word parameter area available in the SVRB. (RTM does not preserve the pointer to this area).

## RESUME

When an operator presses the RESTART key on the processor to break a spin loop, RTM gives control to the FRR established by the routine executing on that processor. In this situation, the first -- and only the first -- FRR to get control can request RESUME. As a result of a RESUME request, RTM terminates the unit of work executing on the processor identified by the recovery routine. Execution resumes with the next sequential instruction on the interrupted processor.

RESUME should be specified only when the main routine is deliberately spinning waiting for a resource to be freed by the another processor.

## RETRY

A retry request from a recovery routine asks RTM to continue execution of the code that established the recovery routine at some appropriate point. The retry routine executes in the same addressing mode as the issuer of the ESTAE macro instruction. The recovery routine cannot change the addressing mode of the retry routine. Note that retry is not always permitted. Whenever the system cannot permit a retry, RTM sets the SDWACLUP bit in the SDWA to one. If a recovery routine requests retry when it is not allowed, RTM ignores the request and continues with termination (percolates).

Any recovery routine that requests a retry must include logic designed to avoid recursion, to prevent the creation of a tight loop between the recovery routine and its retry routine. For example, if the recovery routine supplies a bad retry address to RTM, and the execution of the first instruction at the given address causes a program check, the first recovery routine to get control is the one that just requested the retry. If the recovery routine requests another retry at the same address, the loop is created.

The environment in which the retry routine gets control from an FRR differs from the environment for a retry routine that gets control from an ESTAE-type recovery routine.

## RETRY FROM AN FRR

An FRR can request a valid retry whenever the SDWACLUP bit in the SDWA is set to zero; to request a retry, the FRR must supply a retry address, the entry point of the retry routine. The retry address is the point in the main routine that is to get control in order to continue its processing. In response to a valid retry request, RTM gives control to the retry address that the recovery routine supplies. The retry routine executes as a continuation of the unit of work that encountered the error. Note that RTM does not delete the FRR that requests a retry; the FRR remains valid and can be entered again.

The environment that exists when the retry routine gets control is described in the following topics.

**Registers:** Upon entry to the retry routine, the contents of the registers are the same as the contents of the SDWASRSV field in the SDWA. The FRR that requests the retry manipulates SDWASRSV to set the contents of the registers for the retry routine. Register 15 always contains the retry address.

**Locks:** The status of locks held is the same on entry to the retry routine as it was when the FRR requesting retry completed its processing.

**Disablement:** The retry routine is entered disabled, if, and only if, the FRR requesting the retry returns to RTM disabled. Otherwise, the retry routine is entered enabled.

**Supervisor/Problem Program 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 key zero.

**Cross Memory Environment:** The PKM, PASID, SASID, and mode 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. 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.

The authorization index (AX) is the current one for the PASID of the retry routine.

**SDWA:** When the retry request comes from an FRR, the SDWA is not available to the retry routine.

## Retry From an ESTAE-Type Recovery Routine

An ESTAE-type recovery routine can request a valid retry whenever the SDWACLUP bit in the SDWA is set to zero; to request a retry, the recovery routine must supply a retry address, the entry point of the retry routine. The retry address is the point in the code that established the recovery routine that is to get control in order to continue its processing. In response to a valid retry request, RTM gives control to the retry address that the recovery routine supplies. The retry routine executes as a continuation of the code that established the recovery routine. That is, the retry routine executes under the same RB that established the ESTAE-type recovery, and RTM purges all RBs more current than the retry RB before giving control to the retry routine. Note that ESTAI is an exception; a retry request from a recovery routine established by the ESTAI parameter of the ATTACH macro instruction must execute under a PRB. If there is a PRB whose RBLINK field points to an RTM2 SVRB, that PRB is used. If there is no previous RTM2 SVRB on the queue, then the retry RB is the newest PRB that is older than the oldest non-PRB. If there is no PRB, retry is suppressed.

RTM purges the RB queue to attempt to cancel the effects of partially-executed programs that are at a lower level in the program hierarchy than the program under which the retry occurs. However, the RB purge does not cancel certain effects on the system such as:

- Subtasks created by an RB to be purged
- Resources allocated by the ENQ macro instruction
- DCBs that exist in dynamically-acquired virtual storage

If there are quiesced restorable I/O operations, the retry routine can restore them. RTM supplies a pointer to the purged I/O request list (PIRL). The retry routine can use SVC RESTORE to have the system restore all I/O requests on the PIRL.

Note that RTM does not cancel the ESTAE-type recovery routine that requests a retry; the recovery routine remains valid and can be entered again.

The environment that exists when the retry routine gets control is described in the following topics.

**Registers:** The contents of the general purpose registers upon entry to the retry routine depend on whether or not RTM was able to obtain an SDWA for the ESTAE-type recovery routine.

If RTM could not obtain an SDWA for the recovery routine, the register contents on entry to the retry routine are:

| Register | Contents |
|---|---|
| 0 | A decimal 12. |
| 1 | Address of the user parameter list established using ESTAE or ATTACH with ESTAI. |
| 2 | A pointer to the PIRL, if I/O was quiesced and is restorable; otherwise zero. |
| 14 | Address of supervisor-assisted exit linkage (SVC 3) |
| 15 | Entry point address of the retry routine. |

The contents of all other registers are unpredictable.

If RTM could obtain an SDWA for the recovery routine, the register contents depend on whether or not the recovery routine requested a register update and/or requested that the SDWA be freed. (See the SETRP macro instruction in Volume 2 of this publication for information on the RETREGS parameter used to request a register update and the FRESDWA parameter used to request that RTM free the SDWA.) The register contents are one of the following:

1. If the recovery routine did not request register update and did not request that the SDWA be freed, the register contents on entry to the retry routine are:

   | Register | Contents |
   |---|---|
   | 0 | Zero |
   | 1 | Address of the SDWA |
   | 14 | Address of supervisor-assigned exit linkage (SVC 3) |
   | 15 | Entry point address of the retry routine |

   The contents of all other registers are unpredictable.

2. If the recovery routine did not request update but did request that the SDWA be freed, the register contents on entry to the retry routine are:

   | Register | Contents |
   |---|---|
   | 0 | A decimal 20 |
   | 1 | Address of the user parameter list established using ESTAE or ATTACH with ESTAI |
   | 2 | A pointer to the PIRL, if I/O was quiesced and is restorable; otherwise zero |
   | 14 | Address of supervisor-assisted linkage (SVC 3) |
   | 15 | Entry point address of the retry routine |

   The contents of all other registers are unpredictable.

3. If the recovery routine requests register update, the contents of the 16 general purpose registers on entry to the retry routine are the same as the contents of the 16 words in SDWASRSV. In this case, the recovery routine provides the contents of the registers for the retry routine by updating any or all of the register slots in the SDWASRSV before returning control to RTM with the retry request. If the recovery routine does not also request that the SDWA be freed, it must keep a pointer to the SDWA; this pointer enables the retry routine to reference and subsequently free the SDWA. Note that register 15 does not contain the entry point address of the retry routine unless the recovery routine sets it up that way.

**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 RTM. Otherwise, an SVC error occurs and the retry routine does not get control.

**Disablement:** The retry routine is always entered enabled.

**Supervisor/Problem Program State:** If the recovery routine was established by an ESTAE macro instruction, the retry routine is entered in the state that existed when the macro instruction was issued. If the recovery routine was established by a FESTAE macro instruction, it is entered in supervisor state. If the recovery routine was established by the ESTAI parameter of the ATTACH macro instruction, the retry routine is entered in supervisor state if, and only if, the RBOPSW of the retry RB is in supervisor state and the main routine was authorized at the time of the error. Otherwise, the retry routine is entered in problem program state.

The main routine is considered to be authorized at the time of the error when at least one of the following is true:

1. The program is APF-authorized.
2. The TCBPKF of the task in error is less than 8.
3. The bit TCBFSM of the task in error (in TCBFLGS3 of the TCB) is on, indicating that all RBs for that TCB execute in supervisor state.

**PSW Key**

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

If the recovery routine was established by the ESTAI parameter of the ATTACH macro instruction, 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:

1. The main routine was authorized (as defined earlier under "Supervisor/Problem Program State") at the time of the error.
2. The RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem program state, and the PKM of that RB (XSBKM field in the XSB control block for that RB) does not have authority to keys less than 8.

Otherwise, the PSW key of the retry routine is that of the TCBPKF.

**PSW Key Mask (PKM):** If the recovery routine was established by the ESTAE macro instruction, the retry routine is entered with the PKM that existed when the macro instruction was issued.

If the recovery routine was established by the FESTAE macro instruction, the retry routine is entered with the PKM that existed at the time of the error.

If the recovery routine was established by the ESTAI parameter of the ATTACH macro instruction, the retry routine is entered with the PKM from the XSBKM field in the XSB control block of the retry RB only if one of the following is true:

1. The main routine was authorized (as defined earlier under "Supervisor/Problem Program State") at the time of the error.
2. The RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem program state, and the PKM of that RB does not have authority to keys less than 8.

Otherwise, the PKM of the retry routine only has authority that is equivalent to that of the TCBPKF.

**Authorization Index (AX):** The retry routine is entered with the authorization index that is in effect for the dispatched address space (HOME or PSAAOLD).

**Cross Memory Mode:** The retry routine is entered in primary mode with PASID = SASID = HASID.

**Addressing Mode:** The Retry routine is entered in the same addressing mode that existed when the recovery routine was entered.

**SDWA:** When RTM obtains an SDWA for an ESTAE-type recovery routine, the recovery routine can make the SDWA available to the retry routine. In this case, the retry routine must free the SDWA using the pointer to it and its length. The length appears in the SDWALNTH field, and the subpool in which the SDWA resides appears in the SDWASPID field.

## Percolation

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 main routine and request that RTM continue with termination (percolate). Note that a recovery routine entered with the SDWACLUP bit set to one in the SDWA, indicating that retry is not permitted, has no choice but to percolate. When the recovery routine requests percolation, the previously-established recovery routine gets control. When a retry is not requested and RTM has entered all possible recovery routines, the unit of work (either an SRB or a task) terminates abnormally. Figure 49 shows the decisions RTM makes to determine which recovery routine is to get control in a particular situation.

When a recovery routine requests percolation, it is cancelled; that is, RTM effectively removes that recovery routine from the environment. A cancelled recovery routine is not entered again unless that recovery routine is established again after a retry. There are two types of percolation: percolation for the same unit of work and SRB-to-task percolation.

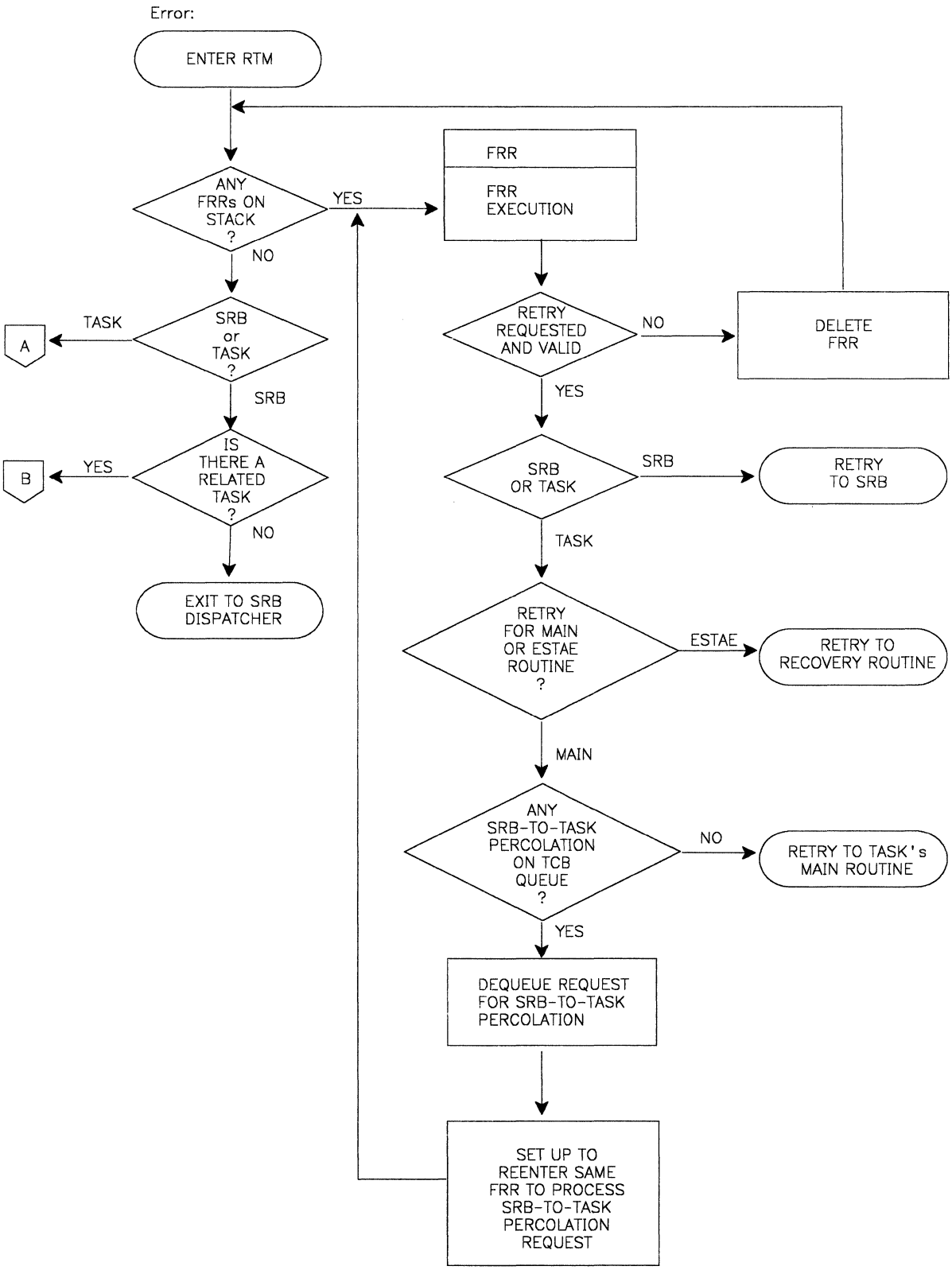



Figure 49 (Part 1 of 3). Routing Control to Recovery Routines

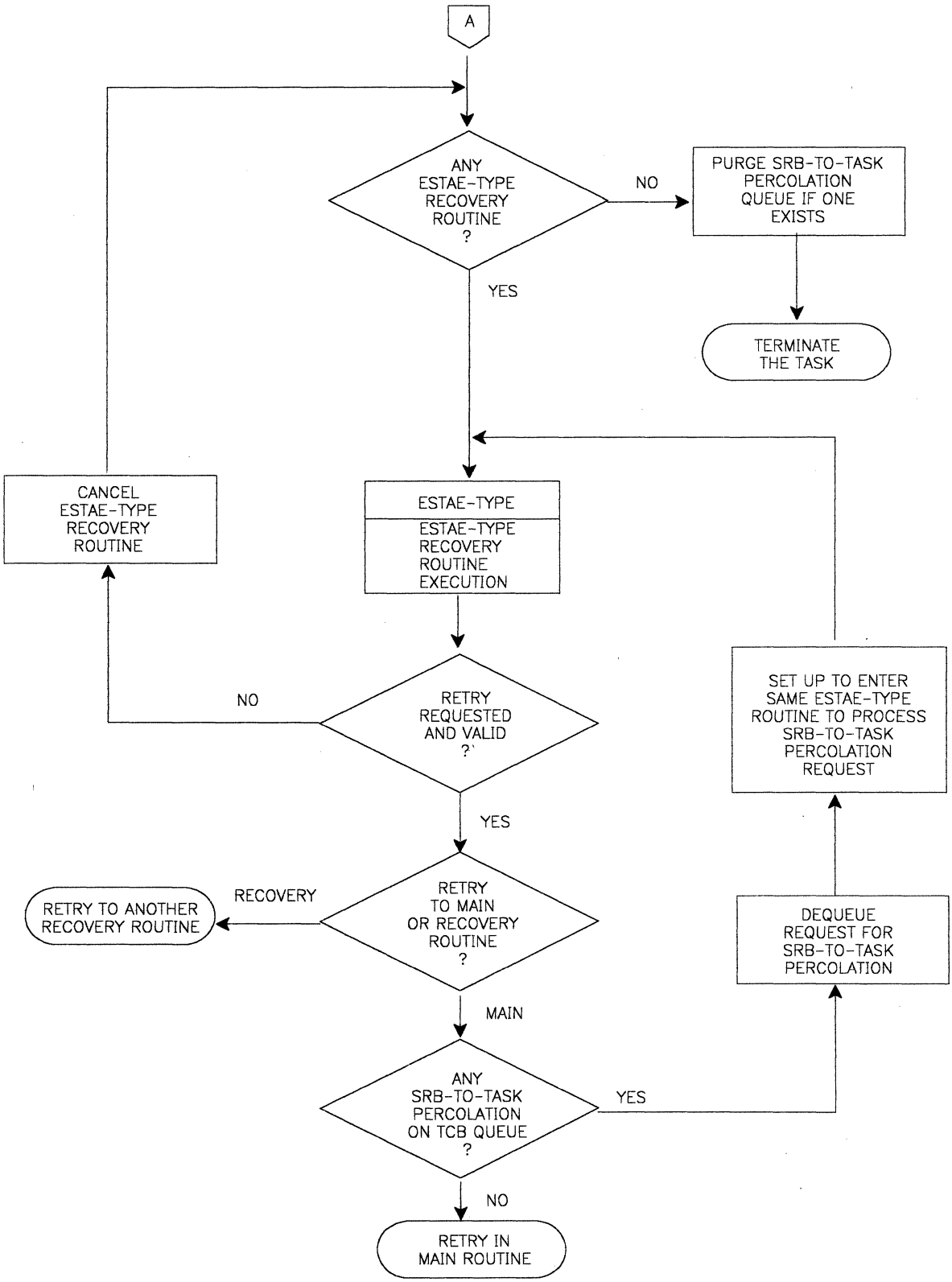



Figure 49 (Part 2 of 3). Routing Control to Recovery Routines

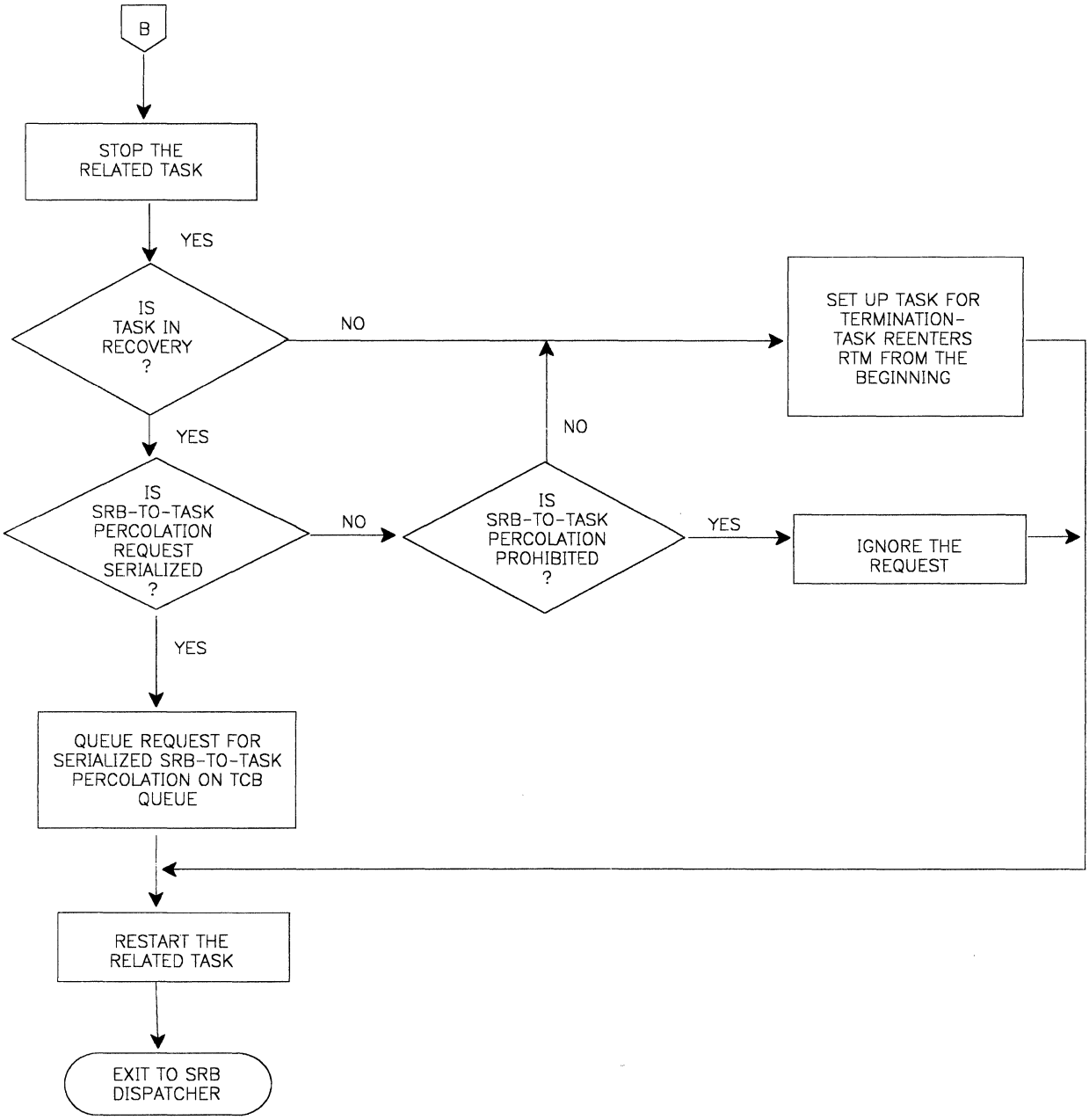



Figure 49 (Part 3 of 3). Routing Control to Recovery Routines

## 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. The environment (except for the cross memory environment) in which a subsequent FRR gets control is the same as the one that existed when the first FRR was entered. The cross memory environment varies because FRRs can turn off the super bits. The lock status, which implies enablement or disablement, could also be different. If an FRR obtains locks that were not held when it was entered and then requests percolation, RTM frees those locks before giving control to the next FRR. Also, if the percolating FRR requested that RTM free any locks, RTM 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 established by the ESTAI parameter of the ATTACH macro instruction can choose either to percolate to a previous ESTAI routine (by setting a return code of 0 for RTM) or to bypass further ESTAI recovery routine processing and continue with termination (by setting a return code of 16 for RTM).

## SRB-TO-TASK Percolation

When an SRB is scheduled and the fields SRBPASID and SRBPTCB are supplied, the task whose TCB address is in SRBPTCB and is executing in the address space whose ASID is in SRBPASID is defined as the SRB's related task. When an SRB with a related task terminates abnormally and the FRR for the SRB does not exist or does not request a retry, the error is percolated to the recovery for the related task. This percolation is called SRB-to-task percolation.

SRB-to-task percolation occurs if none of the FRRs established by the SRB retry or if the SRB does not have an FRR. Either case creates a request for RTM to perform SRB-to-task percolation. RTM ignores the request whenever the related task is terminated. RTM may ignore the request when the related task is already in recovery. If serialization is requested on the SETRP macro instruction in the FRR for that SRB, the percolation request is deferred. (See the SERIAL=YES parameter of the SETRP macro instruction in Volume 2 of this publication.) 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.

RTM processes requests for non-serialized SRB-to-task percolation as follows:

- If the task is in recovery, RTM ignores the request.
- If the task is not in recovery, then RTM abnormally terminates the task and passes the information about the SRB's error to the task's recovery.

RTM processes requests for serialized SRB-to-task percolation as follows:

- If the task is already in recovery, RTM 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, RTM abnormally terminates 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 "Recovery for Recovery" later in this chapter) requests a retry, RTM checks for queued requests for SRB-to-task percolation and takes the following actions before performing the retry:

1. If any requests are queued, RTM dequeues a request and again enters the recovery routine that requested the retry. RTM repeats this process as long as there are queued requests.
2. When the queue is empty or depleted, RTM honors the retry request and gives control to the retry routine.

Figure 49 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."

However, the information in the SDWA describes the error that occurred in the SRB. Whenever serialized SRB-to-task percolation is requested and RTM must queue a request, RTM obtains an area from the user's private area to preserve the information about the SRB's error. If no space is available, RTM cannot preserve that information but still enters the task recovery for the request. If an SDWA is available, RTM sets the SDWARPIV bit to indicate that error-time information is not available. Also, RTM sets the SDWACOMU field to zeroes because RTM cannot preserve its contents to pass from the SRB's FRR to the task recovery routine.

## Recovery for Recovery

In some situations, the function a recovery routine performs is so essential that you should establish 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 establish 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 main 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 in order to pass a particular return code to the main routine. If this recovery routine terminates abnormally and does not establish 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 establish an FRR from either another FRR or from an ESTAE-type recovery routine. You can also establish an ESTAE-type recovery routine from an ESTAE-type recovery routine. However, do not establish an ESTAE-type recovery routine from an FRR because RTM gives control to all FRRs in a recovery path before giving control to any ESTAE-type recovery routines. Therefore, an ESTAE-type recovery routine established in an FRR might not get control in the proper sequence.

Any recovery routine established in a recovery routine is called a nested recovery routine (either a nested FRR or a nested ESTAE recovery routine). Nested ESTAE recovery routines can retry; the retry routine executes under the RB of the ESTAE-type recovery routine that established the nested recovery routine.

Nested FRRs, however, cannot retry. If you need to provide recovery for one FRR with a second FRR, and the second FRR must be able to request a retry, establish both FRRs in the main routine. For example, assume that you want to provide recovery for FRR A with another FRR, FRR B. In the main routine, issue the SETFRR macro instruction for FRR B first, followed by the SETFRR macro instruction that established FRR A. If the main routine terminates abnormally, FRR A gets control. If FRR A then encounters an error, FRR B gets control and, because it is not nested, it can request a retry. Note that RTM deletes FRR A when it gives control to FRR B. If FRR A is needed for the rest of the processing, the retry routine must establish it again. If FRR A is not established again, RTM passes any subsequent errors to FRR B.

All FRRs (nested or not) as well as the main routine, execute under the top RB while in task mode.

## Recovery Routine Guidelines

The actions a recovery routine should take are highly dependent on the function of the main routine; the information presented here summarizes the major decisions -- whether or not you need a recovery routine and what kind of recovery routine you need -- and presents some considerations related to recovery routines that request retry and those that request a dump. "FRR Summary" identifies the information in this book that can help you to design an FRR; "ESTAE-Type Recovery Routine Summary" does the same for ESTAE-type recovery routines.

### Deciding Whether Recovery Is Needed

The first decision you must make is whether or not the main routine requires a recovery routine. Usually, if a function acquires resources that might be requested by another function or that are not known to be related to the task, a recovery routine should be established to free the resources. An example of this type of resource is storage with a subpool that is not task-related (such as subpool 231). Another case that requires a recovery routine occurs when the main routine manipulates such resources as data areas, queues, and data sets that are used by more than one function. The recovery routine in this case should maintain integrity of the resource in case of failure. Recovery routines can also be used to:

- Intercept errors and perform clean-up processing
- Intercept expected program checks and perform the desired action
- Isolate an error to a particular section of processing and continue further processing if possible
- Intercept abends and provide tailored dumps

### Deciding What Type of Recovery Routine to Establish

The second decision is what type of recovery routine to establish. If the function holds a lock, is physically disabled, or is an SRB, an FRR can intercept errors. If the function is running under a task and holds a lock during some portion of its processing, an ESTAE can intercept errors, but the lock is freed before the ESTAE routine gets control. An ESTAE-type recovery routine is useful when losing the locked status can be tolerated, such as when the lock is used only to protect a queue from change while it is being read. Also, if the function is running as an enabled unlocked SRB and there is no need to retry at some point in the SRB, an ESTAE routine associated with its related task could be used to intercept errors in the SRB.

If the function attaches any subtasks, it can also provide recovery for the subtask by specifying the ESTAI parameter on the ATTACH macro instruction.

### Requesting a Retry

A recovery routine that requests a retry should not assume the registers in the SDWA are its own because there are problems, such as errors in called routines that have no recovery or errors in an asynchronous routine (such as an SRB or IRB) that can affect the register contents in the SDWA. The safest method to ensure a successful retry is for the main routine to save volatile information, such as register contents and addresses, in the parameter area passed to the recovery routine and for the recovery routine to use that information for retry. For example, the issuer of the macro instruction can save the base register and data register for the function in the parameter area. This information enables the recovery routine to reference the areas that belong to the function.

### Requesting an ABEND Dump From an FRR

When writing an FRR, note that RTM places the SYSABEND/SYSUDUMP/SYSMDUMP dump options specified on the SETRP macro instruction into the SDWA. Dump options that an FRR specifies replace any dump options that an ABEND macro instruction or a previous recovery routine specified. Also, the CHNGDUMP operator command can add to or override the options. RTM takes one ABEND dump based on the accumulated options. RTM does not take the dump if a retry occurs before the error percolates to an ESTAE-type recovery routine. RTM takes the dump only if all FRRs percolate and no subsequent recovery routine suppresses the dump.

### Requesting an ABEND Dump From an ESTAE-Type Recovery Routine

When writing an ESTAE-type recovery routine, note that RTM accumulates the SYSABEND/SYSUDUMP/SYSMDUMP dump options specified by means of the SETRP macro instruction and places them in the SDWA. During percolation, these options are merged with any dump options specified on an ABEND or CALLRTM macro instruction or by other recovery routines. Also, the CHNGDUMP operator command can add to or override the options. RTM takes one dump as specified by the accumulated options. If the recovery routine requests a retry, RTM takes the dump before the retry. If the recovery routine does not request a retry, RTM percolates through all recovery routines before taking the dump.

### Requesting SVC Dumps and LOGREC Recording From Recovery Routines

By the time an ABEND dump is taken after percolation, valuable function-related information might have been cleaned up by the recovery routines that have already executed. Also, an ABEND dump is returned to the person who ran the abnormally terminating job. Usually the system programming staff wants to see the dumps requested by authorized programs immediately. A recovery routine that is an authorized program can issue the SDUMP macro instruction to request an SVC dump for system programmers to use. Being an authorized program is only one of the rules for requesting SVC dumps. Other restrictions are described in "Using the SDUMP Macro Instruction" later in this section.

Even when SVC dumps are requested, extra care must be taken to preserve valuable function-related information for routines that need to use the branch entry to SVC dump and that cannot wait by using the ECB option on the SDUMP macro instruction.

- For example, information in control blocks used for communication with other address spaces can be changed before a branch entry SVC dump is taken. The recovery routine should use footprint areas to save this information. The available footprint areas include the function's work areas, the variable recording area (VRA) in SDWAVRA, the 4K SQA buffer provided for the SDUMP macro instruction, and any areas the SUMLIST or SUMLISTA parameter specified in an SDUMP macro instruction. Documentation describing how footprint areas are used is very helpful when analyzing the dump or LOGREC entries.

- If volatile information is required, the recovery routine should issue an SDUMP macro instruction after saving the volatile information. Volatile information can be saved by specifying SUMLISTA or SUMLIST=*list address*, BRANCH=YES, and SDATA=SUMDUMP in the SDUMP macro instruction. The areas indicated by the SUMLIST parameter are included in the summary dump. For branch entered SVC dumps, the summary dump is taken by default while the system is disabled. Enabled routines can request SUSPEND=YES on branch entry SVC dump requests in order to preserve pageable volatile data. Volatile data can also be saved by moving it into the 4K SQA buffer described in this section under "Using the SDUMP Macro Instruction," however, this buffer can be used only if no other recovery routine is using it.

Other considerations related to dumping and LOGREC recording are:

- Dumps are usually not required to solve type x37 abends (caused when not enough space is allocated for a data set) and type 913 abends (caused when an operator or user does not supply the correct password). Also, when a prior task is abending and has already taken an ABEND or SVC dump (indicated by the SDWA bits SDWACTS and SDWAMABD being on or the SDWAEAS bit being on), another dump is not necessary. When several recovery routines are written for the same component, the recovery routines that are higher in the hierarchy can test the SDWAEAS bit to check if a lower routine has already taken an SVC dump. (Lower routines should set the SDWAEAS bit to indicate that an SVC dump has been taken.) If other routines have not issued an error message for an abend, the higher recovery routine should issue one.
- The default dump data set is the data set that the user specifies on the SYSABEND, SYSMDUMP, or SYSUDUMP DD statement. If the function executes in key 0, or if it has access to restricted data that should be kept secure in a dump, either the data set specified on the DD statement should be secure or the SDUMP macro instruction should be used to take the dump. The SDUMP macro instruction can include the DCB option for a secure data set.
- When using the SDUMP macro instruction, consider the following:
  - The SDUMP macro instruction should specify a title that summarizes the problem and function. The print dump service aid inserts the first 62 characters of the title on each output page. The title should include at least the name of the module that failed and the name of the recovery module, as in the following example:

    ```
    OC4 ABEND IN OPEN, ERRMOD=IFGORROA,
    ISSUER=IGG020FC, JOBN=C49JACIA, STEPN=GO,
    SDWAVRA=8417F0
    ```

  - The SDUMP macro instruction should not specify all SDATA parameters unless all storage areas are necessary to correct the error. If a particular area is not required, omit the corresponding keyword or do not specify the SDATA option, if there is one. Whenever possible, tailor the SDUMP using the storage list options (STORAGE=, LIST=, SUMLSTA=, SUMLIST=, and LISTA=).
- When LOGREC recording is requested (the default for the first FRR if not specified on the SETRP macro instruction), the name of the module that failed and the name of the recovery module should be saved in the appropriate SDWA fields (SDWAMODN, SDWACSCT, and SDWAREXN). Additional functional information should be saved in the SDWACRC, SDWACID, SDWAMLVL, SDWASC, and SDWARRL fields, and in the SDWA variable recording area (SDWAVRA). For a way to map the contents of the VRA, see the VRADATA macro instruction in Volume 2.

### FRR Summary

If you decide to use an FRR to provide recovery, review the following:

- The syntax of the SETFRR macro instruction in Volume 2.
- The information about FRRs presented under "System Environment" on page 1-154.
- The register use information for FRRs presented under "REGISTER INTERFACE" on page 1-158.
- The information about the SDWA presented under "SYSTEM DIAGNOSTIC WORK AREA (SDWA)" on page 1-159. Note that the SDWA is fully described in the *Debugging Handbook*.
- If your FRR might request a resume or a retry, the information presented under "RESUME" on page 1-166 and "RETRY FROM AN FRR" on page 1-166.
- The syntax of the SETRP macro instruction in Volume 2.
- The information presented under "Recovery Routine Guidelines" on page 1-176.

### ESTAE-Type Recovery Routine Summary

If you decide to use an ESTAE-type recovery routine to provide recovery, review the following:

- If you are using ESTAE, the syntax of the ESTAE macro instruction in Volume 2.
- If you are using FESTAE, the information presented under "Using the FESTAE Macro Instruction" on page 1-164 and the syntax of the FESTAE macro instruction in Volume 2.
- If you are using the ESTAI parameter, the syntax of the ATTACH macro instruction in Volume 2.
- The register use information for ESTAE-type recovery routines presented under "REGISTER INTERFACE" on page 1-158.
- The information about ESTAE-type recovery routines presented under "System Environment" on page 1-154.
- The information about the SDWA presented under "SYSTEM DIAGNOSTIC WORK AREA (SDWA)" on page 1-159. Remember that an SDWA might not always be available to your routine. The SDWA is fully described in *Debugging Handbook*.
- If your ESTAE-type recovery routine might request a retry, the information presented under "Retry From an ESTAE-Type Recovery Routine" on page 1-167.
- The syntax of the SETRP macro instruction in Volume 2.
- If your main routine is a task related to an SRB, the information presented under "SRB-TO-TASK Percolation" on page 1-174.
- The information presented under "Recovery Routine Guidelines" on page 1-176.

### STAE/STAI Exit Routines

The STAE macro instruction causes a recovery routine address to be made known to the control program. This recovery routine is associated with the task and the RB that issued STAE. Use of the STAI option on the ATTACH macro instruction also causes a recovery routine to be made known to the control program, but the routine is associated with the subtask created via 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 exit routine specified by the most recently issued STAE macro instruction gets control and executes 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 instruction with the create (CT) parameter. Each issuance makes the previous STAE environment temporarily inactive. The environment becomes active when the current STAE environment is canceled.

A STAE environment is canceled when the RB that created it goes away (unless it issues XCTL and specified the XCTL=YES parameter on the STAE macro instruction), when the STAE macro instruction is issued with the CANCEL option, or when the STAE routine receives control. If a STAE exit routine receives control and requests retry, the retry routine reissues the STAE macro instruction if it wants continued STAE protection.

A STAI environment is canceled if the task completes or if it requests that termination continue and no further STAI processing be done. In the later case, all STAI exits for the task are canceled.

**Interface to a STAE/STAI Routine:** Prior to entering a STAE/STAI recovery routine, the control program attempts to obtain and initialize a work area that contains information about the error. The first word of the SDWA contains the address of the parameter list specified on the STAE macro instruction or the STAI parameter or the ATTACH macro instruction.

Upon entry to the STAE routine, parameter registers are as follows:

If an SDWA was obtained:

| | | |
|---|---|---|
| Register | 0 | a code indicating the type of I/O processing performed:<br>0 active I/O has been quiesced and is restorable.<br>4 active I/O has been halted and is not restorable.<br>8 no active I/O at ABEND time.<br>16 active I/O, if any, was allowed to continue. |
| Register | 1 | address of the SDWA. |
| Register | 13 | save area address. |
| Register | 14 | return address. |
| Register | 15 | address of STAE exit routine. |

If no SDWA was available:

| | | |
|---|---|---|
| Register | 0 | a code of 12 to indicate that no SDWA was obtained. |
| Register | 1 | ABEND completion code. |
| Register | 2 | address of user-supplied parameter list. |
| Register | 13 | unpredictable. |
| Register | 14 | return address. |
| Register | 15 | address of STAE exit routine. |

When the STAE or STAI routine has completed, it should return to RTM via the contents of register 14. Register 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. |
| 16 | No further STAI/ESTAI processing is to occur. This code may only be issued 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, 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 instruction was issued by a subtask and the attaching task abnormally terminates.
- If the recovery routine was specified for a subtask, via the STAI parameter of the ATTACH macro instruction, and the attaching task abnormally terminates.
- If a problem other than those above arises while RTM is preparing to give control to the STAE routine.
- If another task in the jobstep terminates without the step option.

**STAE/STAI Retry Routines:** If the STAE retry routine is scheduled, the system automatically cancels the active STAE environment; the preceding STAE environment, if one exists, then becomes the active one. Users wanting to maintain STAE protection during retry must reestablish an active STAE environment within the retry routine, or must issue multiple STAE requests prior to the time that the retry routine gains control.

Like the STAE/STAI exit routine, the STAE/STAI retry routine must be in storage when the exit routine determines that retry is to be attempted. If not already resident in your program, the retry routine may be brought into storage via the LOAD macro instruction by either the main program or exit 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 environment that requested retry is canceled and the request block queue is purged up to, but not including, the RB of the program that issued the STAE macro instruction. This is done by pointing each RB old PSW to an SVC 3 (EXIT) instruction. 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 deleted from the I/O restore chain.

The RB purge is an attempt to cancel the effects of partially executed 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. Reason: subtasks cannot be associated with an RB; the structure is defined via TCBs.
- Resources allocated by the ENQ macro instruction. Reason: ENQ resources are associated with the TCB and are not identifiable at the RB level.

- DCBs that exist in dynamically acquired virtual storage. Reason: Only DCBs in the program, as defined by the RB via the CDE itself, are closed.

If there are quiesced restorable input/output operations, they can be restored, in the STAE retry routine, by using word 2 in the SDWA. Word 2 contains the pointer to the purged I/O request list (PIRL) passed as a parameter to SVC Restore. SVC Restore is used to have the system restore all I/O requests on the PIRL.

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

| | | |
|---|---|---|
| Register | 0 | 0 |
| Register | 1 | Address of the SDWA. |
| Register | 2-13 | Unpredictable. |
| Register | 14 | Address of an SVC 3 EXIT instruction. |
| Register | 15 | Address of the STAE/STAI retry routine. |

When the storage is no longer needed, the retry routine should use the FREEMAIN macro instruction to free the first 104 bytes of the work area. 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 instruction. If the retry routine is in the control program key, storage must be freed from subpool 250. The remainder of the work area's storage was freed by RTM during STAE/STAI processing.

If the ABEND/STAE interface routine was not able to obtain storage for the work area, register contents are as follows:

| | | |
|---|---|---|
| Register | 0 | 12 |
| Register | 1 | ABEND completion code. |
| Register | 2 | Address of PIRL or 0 if I/O is not restorable. |
| Register | 14 | Address of an SVC EXIT instruction |
| Register | 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 established or at the time of the error. Otherwise, the retry routine is entered in problem program state.

The task is considered to be authorized at the time the STAE routine is established when at least one of the following is true:

1. The task is APF-authorized.
2. The requestor is in supervisor state.
3. The requestor has a PSW key less than 8.
4. The TCBPKF of the task is less than 8.
5. The PKM of the requestor allows keys less than 8.

The main routine is considered to be authorized at the time of the error when at least one of the following is true:

1. The task is APF-authorized.
2. The TCBPKF of the task in error is less than 8.
3. The bit TCBFSM of the task in error (in TCBFLGS3 of the TCB) is set to one, indicating that all RBs for that TCB execute 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:

1. The task was authorized at the time of the error as described above.
2. The RBOPSW of the retry RB has a key greater than or equal to 8 and is in problem program state, and the PKM of that RB (XSBKM field in the XSB control block for that RB) does not have authority to keys less than 8.

Otherwise, the PSW key of the retry routine is that of the TCBPKF.

## Uses of Resource Managers

A resource manager routine 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. MVS/XA provides resource managers that are invoked to establish routines that "clean up" the queues and control blocks associated with the resources.

In general, MVS/XA does not provide a resource manager for a function that is totally self-contained. A self-contained function does not allocate any resources to the requestor or reclaims all allocated resources before the function returns control. Examples of self-contained functions are EXTRACT and TTIMER. MVS/XA does provide a resource manager for functions that are not self-contained -- that do allocate resources to the requestor. Such services fall into two categories: paired requests (such as OPEN/CLOSE and ENQ/DEQ) and requests that invoke an asynchronous process (such as STIMER and I/O requests).

The system-provided resource managers can, for example, perform clean-up work for a subsystem. The responsibilities of a resource manager include:

- For task termination, removing all traces of the fact that the TCB for the terminating task at one time was connected to, allocated to, or associated with the resource in question. The resource should be left in such a state that it can be reused by another task in the address space or in the system.
- For address space termination, releasing all system queue area and common storage area control blocks obtained for the use of the terminating address space. Also, any buffers, bit settings, pointers, and so on relating to the terminating address space should be reset to make the system appear as if the ASID or ASCB of the terminating address space never existed.

The resource manager is also responsible for establishing a recovery environment when first entered to protect itself against errors during its own processing. For SRBs, the clean-up routine issues the PURGEDQ macro instruction to ensure that all undispatched SRBs are removed from the SRB dispatching queue.

If an installation creates a function that is not totally self-contained, the installation should also provide a resource manager for that function. The installation-created resource manager should perform the same basic tasks for this new function that system resource managers perform for system functions. See *SPL: System Modifications* for information on how to write an installation resource manager.

# Protecting the System

Protecting the system or maintaining system integrity is a major consideration in large systems. This chapter includes information concerning the following topics:

- System integrity
- Using the authorized program facility (APF)
- Using the resource access control facility (RACF)
- Protecting low storage

## System Integrity

System integrity is defined as the ability of the system to protect itself against unauthorized user access to the extent that security controls cannot be compromised. That is, there is no way for an unauthorized problem program using any system interface to bypass store or fetch protection, bypass password checking, bypass RACF checking, or obtain control in an authorized state.

**Note:** An authorized program in MVS/XA is one that executes in a system key (keys 0-7), in supervisor state, or is authorized via the authorized program facility (APF).

### Documentation on System Integrity

This section contains information about MVS/XA system integrity. The related topic of security in regard to the physical environment of a computing system 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*

### 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 control program 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 MVS/XA control program employs to maintain system integrity.

### Elimination of Potential Integrity Exposures

MVS/XA 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 the section "Locking.")

Each of the following descriptions is a guideline to aid the installation in:

- Eliminating that area 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-standard interfaces to the system control program.

There should be no impact on installation-written routines that use standard interfaces (problem program/system interface described in an SRL) because no standard interfaces for system integrity support have been removed from the MVS/XA system control program. However, some routines now require authorization for use.

### User-Supplied Addresses for User Storage Areas

A potential integrity exposure exists whenever a routine having a system protection key (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 MVS/XA convention that system-key routines obtain the protection key of the user before accessing the user-specified area of storage. For example, MVS/XA data management SVC routines (which generally execute in key 5) assume the user's key before modifying a data control block (DCB) or an I/O block (IOB).

### 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, a 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 a privileged state (because such control blocks might contain the addresses of routines that run in privileged mode or with a system (0-7) key).
- 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 MVS/XA, 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. For example, the first load of an SVC can receive control with a valid TCB address 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 and/or data.

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 MVS/XA employs to comply with the requirement:

- In general, authorized program requests to load other authorized programs are satisfied only from authorized system libraries (see the topic "Control Program Extensions" described in this section.)
- MVS/XA 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 the topic "User-Supplied Addresses of Protected Control Blocks" described earlier in this section.)

## 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, in MVS/XA, most SVC B-type 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 MVS/XA routines have simply adopted the convention of assuming the key of their caller before calling the SVC B routine. (For additional information see the section "Writing SVC Routines" later in this book.)

## 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.

MVS/XA maintains such items in system 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 MVS/XA 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 CPU is key 0 fetch-protected from programs attempting to access the PSA while executing on another CPU.

### Control Program Extensions

This potential problem area involves the somewhat hazy distinction that exists between the control program and certain types of problem programs. In most installations, there are problem state/user key (keys 8-15) 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 facility should be used to restrict them and to authorize the control program extensions that use them.

## Using the Authorized Program Facility (APF)

The authorized program facility (APF) is a facility that an installation manager uses to protect the system. In MVS, certain system functions, such as all or part of some SVCs, are sensitive; their use must be restricted to users who are authorized. An authorized program is one that executes in supervisor state, with a PSW key of 0-7, or with APF authorization. In addition, an authorized program should be thoroughly tested so that its use does not compromise the system.

The MVS/XA supervisor uses APF to protect the system as follows:

1. The supervisor limits the use of sensitive system SVC routines and optionally, sensitive user SVC routines, to authorized programs by issuing a TESTAUTH macro instruction before giving control to the SVC routine. TESTAUTH determines, among other things, the authorization status of the calling program. The installation can then use APF to prohibit unauthorized programs from using sensitive SVC routines.
2. The supervisor ensures that all modules in an authorized job step task are fetched only from authorized libraries. The supervisor thus prevents the unauthorized counterfeiting of any module in an authorized job step task's module flow.

### APF Authorization

APF is a mechanism that allows a program to become authorized. APF authorization is established at the job step task level and depends on the authorization status of the first module of the job step task at the time the job step task is initiated. Only when the first module loaded meets both of the following conditions is the job step task marked APF-authorized:

- The module comes from an authorized library.
- The module was link-edited with the authorization code AC = 1. This code is contained in a bit setting in the partitioned data set (PDS) directory entry for the module.

The authorization code is meaningful only when the load module resides in an authorized library and is executed as the first module of a job step task. Thus, even though a program is link-edited with AC = 1, it can run as an authorized program only if it is loaded from an authorized library. When this occurs, the program manager then verifies that all subsequent modules for that program also come from authorized libraries; if they do not, a 306 abend results. Authorized libraries are marked at OPEN time by having a bit turned on in their DEBs.

The names of programs that are APF-authorized in the systems IBM provides can vary from one release to another. To determine which programs are APF-authorized in the current system, list the PDS directories of the authorized libraries (see "Authorized Libraries" later in this discussion) and check for modules that are marked authorized (AC=1). Only modules marked authorized and any modules they invoke from authorized libraries can ever run as authorized programs.

## Authorized Programs

MVS/XA considers a program authorized when that program executes in any one of the following states:

- Supervisor state (bit 15 of the PSW is zero).
- A system key (bits 8-11 of the PSW are in the range 0-7).
- As part of an APF-authorized job step task (bit JSCBAUTH in the JSCB is 1).

However, MVS/XA sometimes distinguishes authorization between a program that runs either in supervisor state or system key and a program that is APF-authorized. For example, the use of certain keywords in some macro instructions is restricted to programs running in supervisor state or system key; programs that are APF-authorized but not in supervisor state or system key cannot use these keywords.

When MVS/XA attaches the first load of a job step task (identified via the JSTCB keyword in the ATTACH macro instruction), program management decides whether to mark the task authorized or to leave it unauthorized. If the first load module has AC=1 and comes from an authorized library, the program is considered APF-authorized, and the task is marked APF-authorized. If these conditions are not met, the task is not marked APF-authorized and cannot normally become so during the life of the job step. (See "Authorization Results Under Various Conditions" for exceptions to this rule.)

You should recognize the distinction between an authorized program and an authorized user. To use restricted functions in MVS/XA a program must be authorized; that is, in supervisor state, running with a system key, or part of an APF-authorized task or some combination of these that satisfies the restriction. Any user, however, can submit a job that executes an authorized program. To restrict a program to an individual user or a class of users, you can use existing data set security facilities to place the program in a library, other than LINKLIB, SVCLIB, or LPALIB, protected by RACF or by a password. If the program is an authorized program, the library must be an authorized library.

**Note:** An authorized program could also be restricted by defining it as a RACF resource and using the RACHECK macro instruction in the program to verify the user's authorization.

## Authorized Libraries

APF-authorized programs must reside in authorized libraries. The authorized libraries in MVS/XA are:

- SYS1.LINKLIB
- SYS1.SVCLIB
- SYS1.LPALIB (only during an IPL; see note 2)
- Installation authorized libraries

To allow an installation to authorize libraries, MVS/XA provides member IEAAPF00 and supports optional members IEAAPFxx in SYS1.PARMLIB. During IPL, the system uses the contents of one member, specified at IPL, to build the APFTABLE, which contains the names of authorized libraries and the serial numbers of the volumes on which they reside. SYS1.LINKLIB and SYS1.SVCLIB are automatically placed in the first two entries of the APFTABLE; the remaining entries contain the names from member IEAAPFxx (where xx is the identifier of the member). The volume serial numbers in the entries prevent the system from

obtaining data from a non-authorized library having the same name as an authorized library but residing on a different volume.

The LNKLSTxx members of SYS1.PARMLIB denote libraries to be concatenated to SYS1.LINKLIB. If libraries in the LNKLST concatenation are accessed through either JOBLIB or STEPLIB DD statements, MVS/XA does not consider them authorized unless the installation has also placed the names of the libraries in IEAAPFxx. As long as a load module is in an authorized library, an authorized program can load it. The installation is responsible for ensuring that duplicate module names are not permitted across authorized libraries to preclude access to an incorrect module resulting in possible integrity exposures.

For details concerning SYS1.PARMLIB members IEAAPFxx and LNKLSTxx, see *Initialization and Tuning*.

**Notes:**

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. After IPL, SYS1.LPALIB is not an authorized library. SYS1.LPALIB is authorized only during NIP processing when the system builds the pageable link pack area (PLPA). All modules in PLPA are marked as coming from an authorized library. SYS1.LPALIB becomes an authorized library after IPL only when an installation places its name in the IEAAPFxx member of SYS1.PARMLIB, and there is no reason to do so.

### Mixing APF and Non-APF Libraries in LNKLST

You can include the LNKAUTH parameter in the system parameter list, IEASYSxx, to indicate whether all data sets in the LNKLST concatenation are to be treated as APF authorized (LNKAUTH = LNKLST) or whether only those that are named in the APFTABLE are to be treated as APF authorized (LNKAUTH = APFTAB). This means that it is possible to mix APF authorized libraries and non-APF authorized libraries in the LNKLST concatenation. The LNKAUTH parameter is also described in *Initialization and Tuning*.

## Using APF

APF allows an MVS/XA installation to restrict use of SVC routines to authorized programs and to restrict access to load modules; that is, APF prevents authorized programs from accessing any load module that is not in an authorized library.

### Restricting The Use of Sensitive Routines

MVS/XA provides the capability to restrict the use of sensitive routines to authorized callers. To restrict the use of an SVC routine, use the TESTAUTH macro or use the SVCTABLE macro during system generation. To restrict the use of non-SVC routines, use the TESTAUTH macro.

**Using the SVCTABLE Macro to Restrict the Use of an SVC:** You can specify the FC01 parameter on the SVCTABLE macro instruction during system generation to restrict sensitive SVCs to authorized callers. MVS ensures that only authorized programs can access routines so restricted. To successfully invoke a restricted SVC routine, the calling program must be in supervisor state, or running under a system key, or APF authorized. If an unauthorized program tries to access a restricted SVC, an 047 abend results.

**Using the TESTAUTH Macro to Restrict the Use of a Routine:** You can use the TESTAUTH macro to restrict the use of any routine, including SVC routines. TESTAUTH enables you to restrict an entire routine or particular paths through the routine when only a portion of the routine's function is sensitive. Though you can specify any combination of system key, supervisor state, and APF authorization for TESTAUTH to test, you should specify only those

conditions that you consider essential. If any of the conditions specified to TESTAUTH are present, TESTAUTH returns an indication that the caller is authorized. For example, to validate the authorization status of programs requesting restricted functions, various system routines use TESTAUTH to make the following distinctions:

1. The caller is executing in supervisor state, system key, or both.
2. The caller is an APF-authorized task (the JSCBAUTH bit in the JSCB is on).

The TESTAUTH macro instruction, inserted at appropriate locations in a routine, returns an authorized or unauthorized indication. The routine can then take appropriate action based upon this indication.

The TESTAUTH macro is not used to control the use of I/O appendages. I/O appendages are controlled by means of the IEAAPP00 member of SYS1.PARMLIB. (See the description of this member in *Initialization and Tuning*.)

## Restricting Load Module Access

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. If an authorized program tries to access a module that is not in an authorized library, a search is done to find a copy of the module in an authorized library. If a copy is found, then processing continues with that copy of the module. If there is not a copy of the module in any authorized library, a 306 abend results.

## Assigning Authorization

An installation can assign load modules the APF-authorization code either through the PARM field on the link edit step or through a linkage editor control statement. The authorization code of a load module has meaning only when it resides on an APF-authorized library and when it is executed as the first program of a job step attach.

To assign an authorization code via JCL, code AC=1 in the operand field of the PARM parameter of the EXEC statement as follows:

```
//LKED   EXEC  PGM=HEWL,PARM='AC=1',...
```

If no authorization code is assigned in the linkage editor step, the default is non-authorization. The authorization code for a given output load module can be overridden with the SETCODE control statement.

The SETCODE statement establishes authorization for a specific output load module. If it is used, you must place it before the NAME statement for the load module. The format of the SETCODE statement is:

```
SETCODE   AC(1)
```

If more than one SETCODE statement is assigned to a given output load module, the last statement found is used.

In the example in Figure 50, 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 50. Assigning Authorization via SETCODE

No security or integrity exposure exists if a program is link-edited into an unauthorized library 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 program is abended.

## Authorization Results Under Various Conditions

When a program issues an SVC or accesses a load module through a LINK, LOAD, or XCTL macro instruction, authorization is straight-forward; the only factors considered are whether the calling program is authorized and whether the called program is a restricted SVC or a load module in an authorized library. Figure 51 summaries the authorization rules.

| Rule | Abend Resulting From Violation |
|---|---|
| 1. An unauthorized routine cannot call a restricted SVC. | 047 |
| 2. A routine running in supervisor state, system key, or APF-authorized cannot call programs residing outside APF-authorized libraries. | 306 |

Figure 51. Authorization Rules

The rules shown in Figure 51 are also true when the ATTACH macro instruction 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 libraries if the following conditions are met:

- The caller is not running APF-authorized or the caller turns authorization (JSCBAUTH) off until program fetch obtains the subtask.
- 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 library and is link edited with the APF-authorized attribute, then the task executes with APF authorization.

Another factor to be considered when using ATTACH is the JSTCB keyword. A routine running in supervisor state or system key can attach a task that is allowed to become APF-authorized if the routine specifies JSTCB=YES.

### Guidelines for Using APF

Installations using APF authorization must control which programs are stored in authorized libraries. 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. To ensure that this assumption is valid, the installation should:

- Ensure that all programs that will run as authorized programs adhere to the installation's integrity guidelines.
- Protect authorized libraries through RACF or passwords to ensure that only selected users can store programs in these libraries.
- Ensure that no two load modules with the same name exist across the set of authorized libraries. 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, become APF-authorized, and thus possibly bypass validity checking or critical logic flow.
- Ensure that IEAAPFxx does not contain the names and volume serial numbers of data sets that no longer exist. If it does, a user could assign his own data sets with the same names on the same volumes and cause his own libraries to become authorized.

## 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.

When users, groups, DASD data sets, tape volumes, DASD volumes, terminals, and other resources are defined to RACF, RACF builds and stores their descriptions in profiles on the RACF data set. RACF uses these profiles for RACHECK authorization checking and RACINIT user identification and verification.

For more information on RACF, see *Resource Access Control Facility (RACF) - General Information Manual.*

### Defining a Resource to RACF (RACDEF)

The RACDEF macro instruction can be used to define, modify, and delete resource profile (for example, a tape volume profile and a DASD data set profile) for RACF.

The resource manager responsible for establishing and maintaining the resources issues the RACDEF macro instruction to define or delete the resource profile.

## Identifying a RACF-Defined User (RACINIT)

The RACINIT macro instruction can be used to determine if a userid is defined to RACF and if the user has supplied a valid password, group name, and operator identification. RACF 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.

## Checking RACF Authorization (RACHECK and FRACHECK)

Two macros, RACHECK and FRACHECK, enable you to determine whether a user is authorized to access a RACF resource.

### RACHECK

RACHECK processing determines if a user is authorized to obtain use of a resource (for example, DASD data set, tape volume, or DASD volume) protected by RACF. When a user requests access to a RACF-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.

RACF performs system authorization checking when a resource manager that controls a RACF-protected resource issues the RACHECK macro instruction before allowing a user access to the resource.

The system programmer using this macro instruction 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 main storage for the resource and that different types of access attempts are or are not to be recorded on the SMF data set.

### FRACHECK

The FRACHECK macro provides a fast-path way to perform a function similar to RACHECK. The FRACHECK macro, however, requires that the profile of the resource being checked be in storage. To build an in-storage profile, issue the RACLIST macro before issuing the FRACHECK macro.

## Retrieving and Encrypting Data (RACXTRT)

The RACXTRT macro instruction can be used for either of two purposes. It can be used to retrieve certain specified fields from a RACF user profile or it can be used to encrypt certain clear-text (readable) data.

## Building In-Storage Profiles (RACLIST)

The RACLIST macro instruction can be used to build in-storage profiles from RACF defined class resources. RACLIST processes only the resources described by class descriptors. Once profiles are brought into main storage by RACLIST, FRACHECK and RACHECK macros can be issued for the resources without requiring access to the RACF data set.

### RACSTAT Macro Instruction

RACSTAT processing determines if RACF is active and optionally determines if RACF protection is in effect for a given resource class. The macro can be used to determine if a resource class is defined to RACF.

## Protecting the Vector Facility

Because all users are authorized, by default, to the Vector Facility, an installation that does not want to limit access to the Vector Facility does not have to take any required action.

An installation that wants to limit access to the Vector Facility can use the following RACF commands to do so:

```
RDEFINE FACILITY IEAVECTOR UACC(NONE)
PERMIT IEAVECTOR CLASS(FACILITY) ID(groupx) ACCESS(READ)
CONNECT usern GROUP(groupx)

                    or

RDEFINE FACILITY IEAVECTOR UACC(READ)
PERMIT IEAVECTOR CLASS(FACILITY) ID(groupy) ACCESS(NONE)
CONNECT usern GROUP(groupy)

                    or

RDEFINE SECDATA CATEGORY ADDMEM(VECTOR)
RDEFINE FACILITY IEAVECTOR UACC(READ) ADDCATEGORY(VECTOR)
ALTUSER usern ADDCATEGORY(VECTOR)
```

In addition, an installation can minimize the overhead of authorization checking by coding a RACF global table entry for the Vector Facility, thus eliminating I/O to the RACF data base. This can be done with the following RACF command:

```
RDEFINE GLOBAL FACILITY ADDMEM(IEAVECTOR/READ)
```

## 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. 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 RACF. 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 RACF is present. If RACF is available in the system, the router passes control to the RACF routine (ICHRFR00) that invokes the appropriate RACF function based on the parameter information and the RACF router table (ICHRFR01), which associates router invocations with RACF functions. The RACF router table is described in *SPL: Resource Access Control Facility (RACF)*. Before it calls the RACF routine, the router calls an optional, user-supplied security processing exit if one has been installed. (See the following topic "MVS Router Exit.")

Control points that issue the RACROUTE macro instruction enter the MVS router in the same key and state as the RACROUTE issuer. Control points that continue to issue the RACF macro instructions go directly to RACF, bypassing the router.

For use on an MVS/XA system, the MVS router exit must be link-edited with AMODE(ANY) and RMODE(24.)

## MVS Router Exit

The MVS router provides an optional installation exit that is invoked whether or not RACF is installed and active on the system. If RACF is not available, the router exit acts as an installation written security processing (or routing) routine. If RACF is available, the exit acts as a RACF preprocessing exit. The installation exit should have an AMODE of ANY and an RMODE of 24.

The only way to invoke the MVS router installation exit is by issuing the RACROUTE macro. The exit is entered via a branch and link macro and thus will execute in the same key and state as the issuer of the RACROUTE macro. The exit must be named ICHRTX00 and must be located in the link pack area (LPA). The router passes the parameter list to the installation exit. In addition, the exit receives the address of a 150-byte work area.

Control points that continue to use the RACF macro instructions do not invoke the installation exit.

On entry to the MVS router exit routine, register 1 contains the address of the following area:

| Offset | Length | Description |
|---|---|---|
| 0 | 4 | **Parameter list address** - points to the MVS router parameter list |
| 4 | 4 | **Work area address** - points to a 150-byte work area that the exit can use |

## Return Codes

### MVS Router Exit Codes

The exit routine returns one of the following return codes in register 15:

| Hex (Decimal) | Meaning |
|---|---|
| 0 (0) | The exit has completed successfully. Control proceeds to the RACF front end routine for further security processing and an invocation of RACF. |
| C8 (200) | The exit has completed successfully. The MVS router translates this return code to a router return code of 0 and returns control to the issuer of the RACROUTE macro, bypassing RACF processing. (See the note below.) |
| CC (204) | The exit has completed successfully. The MVS router translates this return code to a router return code of 4 and returns control to the issuer of the RACROUTE macro, bypassing RACF processing. (See the note below.) |
| D0 (208) | The exit has completed processing. The MVS router translates this return code to a router return code of 8 and returns control to the issuer of the RACROUTE macro, bypassing RACF processing. (See the note below.) |
| Other | If the exit routine sets any return code other than those described above, the MVS router returns control directly to the issuer of the RACROUTE macro and passes the untranslated code as the router return code. The exit routine should place the return and reason code information in the parameter list. Further RACF processing is bypassed. |

**Note:** The installation exit routine is responsible for putting RACF compatible return and reason codes in the first two full words of the parameter list. If the exit routine does not issue a specific reason code, it should issue a zero reason code.

### Simulating a Call to RACF

Normally, a caller, such as DFP, IMS, or JES invokes the MVS router and passes it class, requestor, and subsystem parameters via the RACROUTE exit parameter list. Using those parameters, the MVS router calls the router exit, which then returns to the router with a return code. If the return code is 0, as defined above, the router invokes RACF. RACF reports the results of that invocation to the router by entering return and reason codes in registers 15 and 0 respectively. The router converts the RACF return and reason codes to router return and reason codes and passes them to the caller. The router provides additional information to the caller by placing the unconverted RACF return and reason codes in the first and second words (respectively) of the ROUTER input parameter list.

Instead of invoking RACF processing, your installation may choose to have the MVS router exit respond to the caller's request. If that is the case, you must still provide the caller with the RACF return and and reason codes that it expects to receive. To do so, you must set the router exit return code, as defined above, so that RACF is not invoked. However, you must still simulate the results of a RACF invocation by by coding the exit so it places the RACF return and reason codes in the first and second fullwords (respectively) of the RACROUTE input parameter list.

### MVS Router Parameter List

The MVS router parameter list (mapped by macro ICHSAFP) is generated when the RACROUTE macro is issued and describes the security processing request by providing the request type. If the router installation exit exists, the router passes the parameter list to this exit. If RACF is active, the router uses the request type information to invoke the appropriate RACF function.

| Field Name | Offset | Length | Description |
|---|---|---|---|
| SAFPRRET | 0 (0) | 4 | **Return code** - Defines the RACF or installation exit return code. |
| SAFPRREA | 4 (4) | 4 | **Reason code** - Defines the RACF or installation exit reason code. |
| SAFPPLN | 8 (8) | 2 | **Length** - Defines the length of the SAFP parameter list. |
| | 10 (A) | 2 | **Reserved.** |
| SAFPREQT | 12 (C) | 2 | **Request type** - A binary halfword corresponding to the request type on the RACROUTE macro. The request type and the associated request numbers are listed below.<br>AUTH (RACHECK) - 1 (01)<br>FASTAUTH (FRACHECK) - 2 (02)<br>LIST (RACLIST) - 3 (03)<br>DEFINE (RACDEF) - 4 (04)<br>VERIFY (RACINIT) - 5 (05) |
| | 14 (E) | 2 | **Reserved.** |
| SAFPREQR | 16 (10) | 4 | **Request name address** - Points to an 8-byte character field containing the control point name. |
| SAFPSUBS | 20 (14) | 4 | **Subsystem name address** - Points to an 8-byte character field containing the calling subsystem's name, version, and release level. |
| SAFPWA | 24 (18) | 4 | **SAF work area address** - Points to a 512-byte work area for use by the MVS router and the RACF front end routine. |
| | 28 (1C) | 4 | **Reserved.** |
| | 32 (20) | 4 | **Reserved.** |
| SAFPRACP | 36 (24) | 4 | **Offset** - Contains the (signed) offset from the start of the MVS router parameter list to the RACF parameter list. |

## Interface to the MVS Router (RACROUTE)

The RACROUTE macro instruction is the interface to the MVS router that provides a focal point and a common system interface for all products providing resource control. The MVS router first invokes an optional installation exit and then invokes RACF, if RACF is active and installed on the system.

The RACROUTE macro accepts all valid parameters for any of the RACF macros (RACDEF, RACINIT, RACHECK, RACLIST and FRACHECK) and internally issues the appropriate RACF macro to generate a RACF parameter list. When the RACROUTE macro internally invokes the RACF macros, RACROUTE verifies that only valid parameters have been coded and then passes the parameters to the MVS router. Existing control points that invoke RACF processing via the supervisor call interface can continue to do so or can replace the RACF supervisor calls with the RACROUTE macro.

See the RACROUTE macro in Volume 2 for a description of the return codes.

# Changing System Status (MODESET)

The MODESET macro instruction alters selective fields of the program status word (PSW). 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 protection 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 bit 15 is one, the processor is in the problem state. For problem state, the PKM is changed to reflect the PSW key. When bit 15 is zero, the processor is in the supervisor state.

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:

- Scheduler
- Job entry subsystem
- Real storage management
- Virtual storage management
- System resource management
- Supervisor
- Data management
- Telecommunications access method
- 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.

# Protecting Low Storage (PROTPSA)

The low address protection facility provides protection against altering storage addresses in the range of 0-511. The facility is designed to prevent inadvertent program destruction of storage used by hardware to fetch new PSWs for interruption processing. Low address protection does not apply to the storing of status by the processor, such as old PSWs, logout data, and processor logouts, nor does it apply to the data address in channel commands words (CCWs) and indirect data address words (IDAWs).

The PROTPSA macro instruction is used to disable and enable low address protection. To use this macro, programs must execute in supervisor state with PSW protection key 0, must be physically disabled for I/O and external interrupts, and must not issue any SVCs. The program must not call or transfer control to another program while low address protection is disabled. The protection-disabled window, that part of the program that is executing with low address protection disabled, should be as small as possible.

To insure the proper functioning of the low address protection facility, the contents of control register 0 must be maintained. This requires any program that modifies control register 0, except those using the PROTPSA macro instruction, to place a copy of control register 0 into the field PSACR0SV of the PSA.

Bit 3 of control register 0 is defined as the protection bit. When this bit is zero, low address protection is disabled and stores are permitted. If the bit is set to one, low address protection is enabled and stores are not permitted. When a store in the address range 0-511 is attempted and low address protection is enabled, the content of the storage area addressed by the instruction is not modified. The execution of the current instruction is terminated and a protection exception occurs.

# Exit Routines

This chapter includes the following topics:

- Using asynchronous exit routines
- Establishing a timer disabled exit routine

Additional information on exit routines can be found in *SPL: User Exits.*

## Using Asynchronous Exit Routines

An authorized user can request an asynchronous exit routine to execute on behalf of a specific task. Before execution, the exit routine must complete three system control stages, each stage carried out by an individual "exit effector" routine.

The stage 1 exit effector routine creates and initializes an interrupt request block (IRB) that identifies the user's asynchronous exit routine to the system. Interface to the stage 1 exit effector is through the create interrupt request block (CIRB) macro instruction. Volume 2 describes the CIRB macro instruction.

The stage 2 exit effector routine schedules the user's exit routine for execution. Input to the stage 2 exit effector is an interrupt queue element (IQE) initialized by the caller. The IQE identifies the task the exit routine is to execute under and the associated IRB, and it also contains information about the exit routine's characteristics. At the conclusion of stage 2 exit effector processing, the user's exit routine is logically ready for system dispatch and execution. The exit routine is executed only once each time the caller invokes the stage 2 exit effector for that exit routine. Reuse of previously defined control blocks is possible, however. Thus, two or more invocations of the exit routine can use one interface to the stage 1 exit effector.

The MVS/XA dispatcher invokes the stage 3 exit effector to queue the exit routine's IRB to the specified task. The exit routine will then execute on the next dispatch of that task (provided no other IRBs have been scheduled for that task).

### Stage 1 Initialization

The CIRB macro instruction provides the interface to the stage 1 exit effector and is normally a type 1 SVC interface. Also, a branch entry interface is available by specifying BRANCH = YES as one of the macro options. Detailed information about using the CIRB macro instruction is in Volume 2. When invoked, the stage 1 exit effector routine obtains the storage for the IRB and, optionally, the IQE and a problem program work area. Also, the stage 1 exit effector initializes those fields in the IRB necessary to control the execution of the exit routine. At completion of the stage 1 exit effector, the IRB address is returned to the caller in register 1. The IQE, if requested, occupies storage contiguous to the IRB and is pointed to by the RBNEXAV word 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 52.

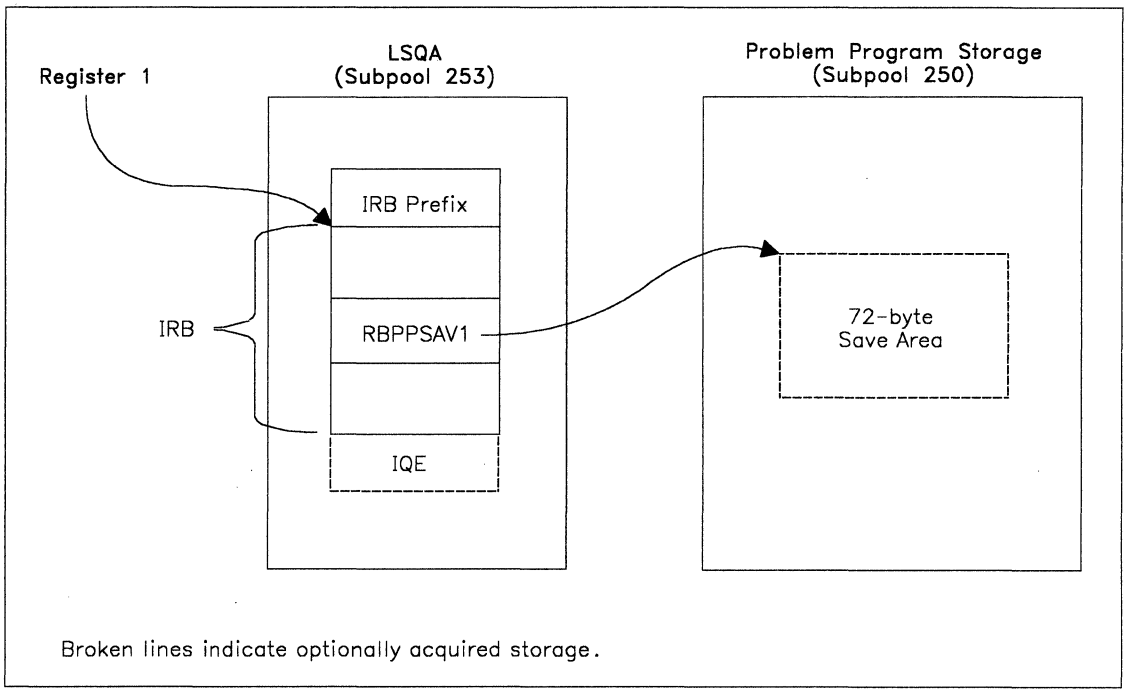



Figure 52. Asynchronous Exit Data Area Configuration

The fields in the IRB initialized by the stage 1 exit effector 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 I/O supervisor uses RQEs.
- 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 (TCBPKF) of the caller if KEY=PP is specified on CIRB.
  - Mode: Supervisor state if MODE=SUPR on the CIRB macro instruction; problem program state if MODE=PP on CIRB.
- RBSIZE -- the size of the IRB (including the size of the IQE if the CIRB specification included the WKAREA parameter).
- RBNEXAV -- the address of the IQE if WKAREA was specified (occupies the first four bytes of the work area).
- RBPPSAV1 -- the address of the problem program save area if SVAREA was specified.

## Stage 2 Scheduling

The user must initialize the IQE to define to the system the task under which the exit routine is to execute. 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 stage 1 exit effector routine.
- IQETCB -- address of the TCB for the task under which the user's exit routine is to execute.

When IRB/IQE initialization is complete, the user should invoke the stage 2 exit effector routine to queue the request (IQE) to the appropriate system asynchronous exit queue. The entry to stage 2 is by branch only, where the branch entry point address is found in the communications vector table (CVT) field CVT0EF00. The interface to the stage 2 exit effector is defined as follows:

| Register | Contents |
|---|---|
| 0 | Irrelevant for scheduling an exit via IQE. |
| 1 | Twos-complement IQE address. |
| 2-13 | Irrelevant. |
| 14 | Return address. |
| 15 | Irrelevant. |

**Note:** Upon return, registers 0, 2-9 and 11-14 are unchanged, register 1 contains a true (non-complemented) IQE address, and register 10 is destroyed.

The caller of stage 2 must:

- Hold the local lock.
- Be addressable in the address space in which the exit routine is to be dispatched.
- Be in supervisor state under protection key zero.

## Stage 3 Execution

Once scheduled by stage 2, the user's exit routine is logically ready for dispatch. Stage 3, effectively a subroutine of the MVS/XA dispatcher, is called to queue the IRB associated with the user's exit to the task the IQE indicates. The user's exit executes as a result of the next dispatch of the task unless a subsequent IRB has been scheduled for the same task. In this case, the second exit routine might be executed first. Stage 3 execution depends on valid information being in the IQE fields. Because stage 3 performs no validity checks on IQE initialization, the user must ensure that the IQE fields are correctly initialized.

### Execution and Termination Characteristics

The following characteristics of an asynchronous exit routine can influence the user's choice of CIRB options and should be considered:

- The exit routine executes as an IRB under the TCB defined by the IQE passed to the stage 3 exit effector routine.
- The exit routine executes enabled in the key and state requested by the CIRB macro instruction interface to the stage 1 exit effector routine.

- Register contents upon entry to the exit routine are:

| Register | Contents |
|---|---|
| 0 | IQE address. |
| 1 | Parameter list address (IQEPARAM). |
| 13 | Problem program register save area address (if any). |
| 14 | Return address (CVTEXIT). |

- Upon termination of the asynchronous exit routine:
  - The IQE is returned to a "next available" queue anchored by the RBNEXAV field in the IRB if the user specified the WKAREA and RETRN=YES options on the CIRB macro instruction. This allows subsequent invocations of the exit routine without requiring the user to repeat requests for stage 1 processing (data area setup).
  - If the user specified the SVAREA and STAB=(DYN) options on the CIRB macro instruction, the problem program register save area is freed.
  - If the user specified the STAB=(DYN) option of the CIRB macro instruction, the IRB and IQE are freed.

## Establishing a Timer Disabled Interrupt Exit

Timer supervision 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 (IEAVRT02). When the time interval expires, the user's DIE routine gains control as an extension of the timer second level interrupt handler (IEAVRTI0). 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.

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. The IHATQE macro is available in macro library APVTMACS. 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").

  TQEVAL -- the desired real time interval (a 64 bit unsigned binary number with bit 51 = 1 microsecond).

  TQEAMODE bit in TQEFLGS3 field -- set to 1 to 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 in the TQEFLGS3 field 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. |

**Note:** The set DIE service routine obtains the dispatcher lock if the caller does not already hold it. After completing its processing, the set DIE routine releases the lock if the caller did not previously hold it. The caller must not hold any lock higher in the locking hierarchy than the dispatcher lock.

The set DIE service routine does not establish its own recovery routine. Any system 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 executing out of the timer SLIH gains control under timer supervision'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. When the timer supervision FRR gets control, it tries to repair any damage to the system's real time queue and then percolates.

## DIE Characteristics

Entry to the DIE routine is in supervisor state, with PSW key zero, disabled, with no disabled global spin locks held. Register contents upon entry are as follows:

- 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.
- Register 2 - 12 are as they were upon entry to the set DIE service routine (or as changed by a previous DIE entry -- "DIE Execution").
- Register 14 contains the return address.

- Register 15 contains the entry point of the DIE routine.
- The contents of floating point registers are unpredictable.

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 must release all locks it obtained but need not save the general purpose registers. Floating point registers, however, must have the same contents on exit as on entry to the DIE.

**DIE Execution:** must be like the execution of an interrupt handler because it executes as an extension of the timer SLIH. 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 cannot reference any private storage areas.
- The DIE must execute disabled. Hence, it cannot cause a page fault.
- The DIE cannot request a local lock or the CMS lock because these are suspend locks and might therefore already be in use. Furthermore, the DIE routine cannot assume whether or not these locks are held upon entry.
- The DIE cannot 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 hold the dispatcher lock upon entry to the TQE ENQUEUE routine.

The input environment for the TQE ENQUEUE routine must be as follows:

- Supervisor state, key zero, and holding the dispatcher lock.
- 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 in TQEFLGS3 field -- 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 in the TQEFLGS3 field 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 registers 2 - 12 are to be changed for the subsequent DIE routine entry, the new values should be set in this eleven 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.

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 (including important flag bits) 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.

**Notes:**

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 the section "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. Timer supervision serializes the use of the real time queue by means of the dispatcher lock.

**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 timer SLIH 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, and with the dispatcher lock. 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.

# User-Written SVC Routines

This chapter includes the following topics:

- Writing SVC routines
- Inserting SVC routines into the control program
- Subsystem SVC Screening

## 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 MVS/XA. Five types of SVC routines are supplied with MVS/XA, 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 your routines must execute serially with respect to other parts of the control program, then you must use the same locking conventions as the control program. If you write two or more SVC routines that must serialize with each other, use the locking facilities or the ENQ and DEQ macro instructions.

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 disabled spin 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.

### 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, because the instruction path lengths for receiving and releasing control are shorter, 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 has the same limitations that apply to the FLIH. When a type 1 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 following register conditions:

| T6EXIT Option | Register 14 | Register 0 | Register 1 | Register 15 |
|---|---|---|---|---|
| CALLER | As on entry | Returned to caller | Returned to caller | Returned to caller |
| DISPATCH | CVTT6SVC | N/A | 0 | N/A |
| SRB | CVTT6SVC | N/A | SRB address | N/A |
| BR 14 | As on entry | Returned to caller | Returned to caller | Returned to caller |

If a type 6 SVC uses the RETURN=SRB exit option on the T6EXIT macro instruction, 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 disabled, it has exclusive use of the processor. Thus, the SVRB save areas are unavailable to a type 6 SVC routine, although the PSA areas can be used instead. When a type 6 SVC is executing, no other task-related activity can occur concurrently. To indicate this situation, the TCBACTIV flag is set. 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 instruction.

### Non-Preemptable SVC routines

You can 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 because the exit function always resets the non-preemptable indicator in the TCB associated with the SVC. This action causes the issuing SVC to lose its non-preemptable state. If a non-preemptable SVC issues a STAX DEFER=NO macro instruction, the SVC routine remains non-preemptable until it exits.

## Programming Conventions for SVC Routines

Figure 53 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 right most column of the figure correspond to the reference notes. If a reference note for a convention does not pertain to a specific type of SVC routine, that type is indicated by an asterisk.

| Conventions | Type 1 | Type 2 | Type 3 | Type 4 | Type 6 | Reference Code |
|---|---|---|---|---|---|---|
| Part of resident control program | Yes | Yes | No | No | Yes | |
| 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 | | | | | 5 |
| Supervisor request block (SVRB) size | No SVRB exists | 224 | 224 | 224 | No SVRB exists | 6 |
| May issue WAIT macro instruction | No | Yes | Yes | Yes | No | 7 |
| May suspend their caller | Yes | No | No | No | Yes | 8 |
| May issue XCTL macro instruction | 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 | | | Not Applicable | 11 |
| Exit from SVC routine | Branch using return register 14 | | | | T6 EXIT or BR 14 | 12 |
| Method of abnormal | ABEND | ABEND | | | ABEND | |
| Recovery | FRR | ESTAE or FRR | | | FRR | 13 |

Figure 53. Programming Conventions for SVC Routines

| Reference Code | SVC Routine Types | Reference Notes |
|---|---|---|
| 1 | all | If your SVC routine is to be reenterable, you cannot use macro instructions whose expansions store information into an inline parameter list. |
| 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:<br>* 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 SVCTABLE macro instruction or may be requested dynamically within the SVC routine.<br>* Types 2, 3, and 4 may also request locks via the SVCTABLE macro instruction or may obtain them dynamically.<br>* 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 lock be held.<br>* 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.<br>* If disabled spin locks are held or obtained, the SVC routine executes in supervisor state, key zero, disabled mode. No SVCs may be issued.<br>* SVCs may not take disabled page faults. Therefore, if a disabled spin 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.<br>* An FRR may be defined for any SVC routine that holds or obtains locks to provide for abnormal termination (see reference code 9).<br>* Type 6 may not request any locks. |
| 4 | all | You must use the following conventions when naming SVC routines:<br>* Types 1, 2, and 6 must be named IGCnnn; nnn is the decimal number of the SVC routine. You must specify this name in an ENTRY, CSECT, or START instruction.<br>* Types 3 and 4 must be named IGC00nnn; nnn is the signed decimal number of the SVC routine.<br>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:<br>* A type 3 SVC identifies a function that is contained in a single load module.<br>* 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 registers is as follows:<br>* Register 0 and 1 contain the same information as when the SVC routine was called.<br>* Register 2 contains unpredictable information.<br>* Register 3 contains the starting address of the communication vector table (CVT).<br>* Register 4 contains the address of the task control block (TCB) of the task that called the SVC routine.<br>* 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.<br>* Register 6 contains the entry point address.<br>* Register 7 contains the address of the address space control block (ASCB).<br>* Registers 8 through 12 contain unpredictable information.<br>* Register 13 contains the same information as when the SVC routine was called.<br>* Register 14 contains the return address.<br>* Register 15 contains the same information as when the SVC routine was called.<br>You must use register 0, 1, and 15 if you want to pass information to the calling program. The contents of registers 2 through 14 are restored when control is returned to the calling program. |
| 6 | 2,3,4 | When a type 2, 3, or 4 SVC routine receives control, register 5 contains the address of the SVRB with in this 224-byte area. This SVRB contains a 48-byte "extended save area." In addition, an area is provided for a STAE control block (SCB); this SCB is used by the FESTAE macro instruction. |
| 7 | 2,3,4 | You can issue the WAIT macro instruction if you hold no locks. You can issue WAIT macro instructions 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 macro instruction in a routine under control of a type 2, 3, or 4 SVRB, the new load module must be located in the fixed or pageable link pack area. 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. |
| 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 reobtained. |

| Reference Code | SVC Routine Types | Reference Notes |
|---|---|---|
| 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, SVC 3 results 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 instruction). 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.<br>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.<br>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 macro instructions). |

## Inserting SVC Routines Into the Control Program

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 and SYS1.LPALIB. When the system is IPLed, the NIP program translates the SVC definitions that you placed on SYS1.PARMLIB into SVC table entries. Then NIP places the SVC table in the extended read-only nucleus.

When NIP processes SYS1.PARMLIB, it 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 *SPL: Initialization and Tuning* 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 cannot modify SVCs that are in the range of 0-199. When you define an SVC with an SVCPARM statement, you define its type as type 1 through type 6, excluding type 5. 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.

## Modifying the SVC Table at Execution Time (SVCUPDTE)

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.

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 SVCPARM statements or the SVCUPDTE macro.

### 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.

## 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 SVC first level interrupt handler (FLIH) determines, for each SVC issued by that task, whether the task can request that SVC function. If the SVC request is invalid, the SVC FLIH gives control to a special error subroutine supplied by the routine that activated the screening function.

The subsystem, executing under PSW protection 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 a 264 byte area called the subsystem screen table. 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)

| Bytes | Content |
|---|---|
| 0-3 | 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. |
| 4 | X'00' -- means that the subroutine is to execute as a Type 1 SVC<br>X'80' -- means that the subroutine is to execute as a Type 2 SVC<br>X'C0' -- means that the subroutine is to execute as a Type 3 or Type 4 SVC<br>X'20' -- means that the subroutine is to execute as a Type 6 SVC |
| 5 | Zero. |
| 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:<br>Bit Lock<br>0 LOCAL<br>1 CMS<br>2 SRM<br>3 SALLOC<br>4 Dispatcher<br>Bits 5-15 are always zero (off). |

2) SSTMASK -- SVC screening mask (256 bytes)

| 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. |

The subsystem must get and initialize the subsystem screen table properly. The subsystem must also free the table before terminating.

# UCB Scan Services

The UCB scan routine (IOSVSUCB) 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 IOSVSUCB you can, for example, find the UCB currently associated with a particular VOLSER or find all tape devices currently allocated.

IOSVSUCB runs in the caller's key, state, and addressing mode. The caller can be in either task or SRB mode with the following restrictions:

- If in task mode, the caller must be enabled and must hold no locks.
- If in SRB mode, the caller cannot hold the UCB lock or any lock higher than that in the locking hierarchy.

## Invoking IOSVSUCB

IOSVSUCB is more general than the IOSLOOK macro instruction, which requires the user to be in supervisor state and to provide the device address as input. See Volume 2 for a description of the IOSLOOK macro.

Each time that you invoke IOSVSUCB, you will obtain the address of the common segment of one unit control block (UCB). In order to scan several UCBs, you must invoke IOSVSUCB repeatedly, once for each UCB. IOSVSUCB 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 you must clear this work area to binary zero. The zeros indicate that IOSVSUCB 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, you must not change the work area. See Figure 56 for an example of how to use IOSVSUCB.

### Input to IOSVSUCB

To use IOSVSUCB, the caller must:

- Obtain a 100-byte work area that starts on a double-word boundary and clear the work area to binary zero each time that the scan is to start with the first UCB in the system or the first UCB in a device class.
- Build a parameter list as shown in Figure 54.

| |
|---|
| 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 IOSVSUCB 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. |

Figure 54. Parameter List for the UCB Scan Routine (IOSVSUCB)

- 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 (IOSVSUCB) (The CVTUCBSC field in the CVT contains the entry point address.) |

**Note:** The data areas that the caller passes to IOSVSUCB 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. If the program runs in 24-bit addressing mode, 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 55 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 55. Device Classes

## Output from IOSVSUCB

When IOSVSUCB returns, register 15 contains one of the following return codes:

| Return Code | Meaning |
|---|---|
| 00 | IOSVSUCB stored a UCB address in the location specified in the third word of the parameter list. |
| 04 | There are no more UCBs. IOSVSUCB set the 100-byte work area to binary zeros. |

**Notes:**

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, IOSVSUCB returns only one UCB address for those devices. Devices with multiple exposures have one UCB associated with each exposure. Therefore, IOSVSUCB returns one UCB address for each exposure.

## Example Using IOSVSUCB

Figure 56 contains an example of how to use IOSVSUCB to find the UCB currently associated with a particular VOLSER. The search is limited to direct access UCBs.

```
FINDVOL  CSECT
PROLOG   STM     R14,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
```

Figure 56 (Part 1 of 2). Example of the UCB Scan Routine (IOSVSUCB)

```
         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
```

Figure 56 (Part 2 of 2). Example of the UCB Scan Routine (IOSVSUCB)

## Obtaining Information from the Input/Output Supervisor (IOS)

The IOSINFO macro instruction obtains the subchannel number for a specified unit control block (UCB) from the input/output supervisor (IOS) without being dependent on the location or format of the information as it is maintained by IOS. 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, IOS 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 via a return code. If the UCB is disassociated from the subchannel after the IOSINFO macro invocation, IOSINFO can not notify the caller. For detailed information on how IOSINFO deals with the disassociation of the subchannel from the UCB, see the instructions for coding the macro in Volume 2 of this book.

# Dynamic Allocation

The allocation of resources performed in response to JCL at step allocation (or at logon, for time-sharing users) can be altered prior to step deallocation (or logoff) by invoking dynamic allocation functions. A job's device requirements might not be fully evident prior to execution; using dynamic allocation, the program can acquire resources as the need develops. Similarly, each job can use common resources more efficiently with dynamic allocation: the resources can be acquired just before use and/or released immediately after use.

You request dynamic allocation functions by invoking SVC 99.

"Dynamic Allocation" introduces the functions available through SVC 99, along with some concepts and processing features that are special to SVC 99.

In addition, this topic describes installation options you can use to control the processing of all SVC 99 requests.

"Requesting SVC 99 Functions" describes how to request SVC 99 functions, including details on coding the SVC 99 parameter list and information on the return codes issued by SVC 99 functions.

"Requesting SVC 99 Functions" also presents all the SVC 99 text unit keys, in numerical order by key within verb code groups, in reference format. In addition, "Requesting SVC 99 Functions" includes a detailed discussion of the processing involved in dsname allocation, and an example of a dynamic allocation request.

The term "resource" means a ddname-data set combination, with any attendant volumes and devices.

"Dynamic allocation functions" refers to the allocation of I/O resources *during program execution*, and all of the related functions of resource allocation performed by SVC 99.

To avoid confusion between the specific function of dynamically allocating a resource and the set of functions provided by the dynamic allocation routines, the first is called *dynamic allocation* and the second *SVC 99 functions*, throughout the remainder of the book.

**Note:** Throughout the remainder of the book, 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 an SVC 99 function – without causing confusion, the words "unallocate/unallocation" have been retained.

## Introduction to SVC 99 Functions

When you invoke SVC 99, you can request five different functions. They are:

- Dynamic allocation — acquiring a resource
- Dynamic unallocation — deallocating a resource
- Dynamic concatenation — associating acquired data sets
- Dynamic deconcatenation — separating associated data sets
- Dynamic information retrieval — obtaining certain data set information

A typical use for SVC 99 functions is in a program that needs temporary use of a volume for which there is heavy contention. In such a case, dynamic allocation and dynamic unallocation provide the means for a program to tie up the volume for only as long as necessary rather than for the total execution time of the program.

Another common use for SVC 99 functions is in a program whose need for I/O resources varies according to the input. Dynamic allocation and dynamic unallocation permit such programs to allocate and then free only the files necessary to process the input, so the specific resources supporting the required files can be in use for the minimum time.

You request SVC 99 functions via the DYNALLOC macro. Instead of specifying operands on the macro, you supply information in the SVC 99 parameter list.

You request a specific SVC 99 function via information in two fields of the parameter list:

- **Verb code** — a one-byte hexadecimal code that describes the function being requested.
- **Text Unit Key** — a two-byte hexadecimal value that describes the processing being requested from the SVC 99 function routine. A number of keys are available for each verb code.

For example, verb code 01 with the key of 2 requests that a new data set be allocated, while verb code 02 with key 7 requests that an existing data set be deallocated.

You use other fields in the SVC 99 parameter list to supply the information required to process your request. See Figure 58 for the entire SVC 99 parameter list.

# Concepts Needed to Understand SVC 99 Processing

SVC 99 processing involves some features dictated by the environment in which dynamic allocation or deallocation takes place, and by the very fact that the SVC 99 function is performed dynamically – while a program is executing. Because SVC 99 was developed to speed up time-sharing operations, these features might appear to be relevant only to a program running in an interactive environment. However, any program that invokes SVC 99 makes use of the features.

An understanding of the features – and the concepts behind them – is essential to an understanding of the SVC 99 functions.

## Processing Control Features

SVC 99 routines provide some controls designed specifically for the unpredictable interactive environment.

Time-sharing command processors use SVC 99 functions to dynamically allocate data sets required for their own processing (for example, work areas), in addition to the data sets the user requests via the time-sharing commands. Because the same command processor can be called again, and different command processors might need the same data sets, the command processors do not deallocate the data sets they allocate for their own use. This avoids allocation processing that would be required when a subsequent command processor requested the same data sets. However, keeping all data sets allocated until the end of a terminal session can also tie up resources that might no longer be needed. The following features help to avoid tying up resources that are not being used:

### In-Use Bit and Attribute

An **in-use bit** for each data set is located in the data set association block (DSAB).

The in-use bit is turned on when a data set is dynamically allocated. In a time-sharing environment, the terminal monitor program (TMP) turns off the in-use bits of all data sets that were dynamically allocated by a command processor when that command processor completes execution. (Turning off the in-use bit does **not** deallocate the data set.) If a subsequent command processor dynamically requests a previously-allocated data set, the in-use bit is turned on again until the command processor completes execution and returns control to the TMP.

As part of the in-use feature, the system also keeps track of the data sets that have been "not-in-use" for the longest time.

### Control Limit

The **control limit** limits the number of data sets that can be allocated but marked "not-in-use" (that is, the in-use bit is turned off).

This control limit is determined by the JCL parameter DYNAMNBR on the EXEC statement in the logon procedure and the number of DD statements in the logon procedure. If the control limit is exceeded when the SVC 99 routines receive a request for a new dynamic allocation, the routines automatically attempt to deallocate enough data sets to meet the control limit, starting with eligible data sets 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.

**Note:** The control limit would not seem to apply to a program running in a batch environment. However, some utilities (IDCAMS, for example) may expect a control limit, and could cause an 043C error reason code from SVC 99 (see Figure 62). If a batch program requires a large number of allocations, you might need to include the DYNAMNBR parameter on the EXEC statement, specifying a large value. The value that you specify must not exceed the maximum number of single unit DD statements allowed for the current TIOT size.

### Permanently Allocated Attribute

The **permanently allocated attribute** prevents the SVC 99 routines from automatically deallocating a particular data set to meet the control limit. The effect of this attribute is to determine a data set's eligibility for automatic deallocation.

The permanently allocated attribute is automatically assigned to data sets allocated through JCL and the ALLOCATE command. In addition, you can request (via the SVC 99 parameter list) that a data set be assigned this attribute when you dynamically allocate the data set.

Here is an example of the use of the permanently allocated attribute: The ATTRIBUTE command processor assigns the permanently allocated attribute to a dummy data set that is associated with other attributes specified on the command. The dummy data set must not be automatically deallocated if the control limit is exceeded; it must remain available, to be referenced by a USING keyword on an ALLOCATE command, until the user FREEs the data set. Therefore, it needs the permanently allocated attribute.

**Note:** Because permanently-allocated resources are not automatically deallocated, and all resources allocated via the ALLOCATE command and JCL are permanently allocated, the control limit primarily limits the number of resources that a terminal user can have allocated at the same time. If no control limit is established, the limit is the maximum number of single-unit DD statements allowed for the TIOT size specified at IPL. For information on how the TIOT size is changed, see "Installation Options For SVC 99 Functions" on page 1-237.

Because a batch application is more predictable than a time-sharing terminal session, the programmer who is using SVC 99 functions in a batch application probably will not need to be concerned with the in-use bit, the control limit, or the permanently allocated attribute. For an exception, see the note under "Control Limit."

### Convertible Attribute

Because a data set requested by a command processor or an application program might already be allocated, the dynamic allocation routines first check for an existing allocation that matches the current request. This avoids redundant allocation processing. In some cases, an existing allocation matches the current request except for some parameters. The dynamic allocation routines can change certain unmatching parameters of the existing allocation to meet the current request if the existing allocation has the **convertible attribute**. The convertible attribute allows the dynamic allocation routines to change the following parameters of the existing allocation:

- Ddname
- Member name
- Status
- Normal and conditional dispositions
- Space
- Deallocation at CLOSE
- Input only
- Output only
- DCB attributes
- Password

In addition, the routines can change the permanently allocated attribute.

The convertible attribute is automatically assigned to all data sets 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.

Although the setting and use of the convertible attribute are transparent to a program running in a batch environment, the SVC 99 routines make use of this feature. All the resources dynamically allocated on behalf of a program running in a batch environment are automatically assigned the convertible attribute. When, for example, a program in your batch job dynamically allocates a data set for input only and later allocates the same data set for output only, the dynamic allocation routines will automatically avoid redundant allocation processing by using the first allocation and changing "input only" to "output only."

# Functions Available Through SVC 99

When you invoke SVC 99, you request that its routines perform one of five general functions. The following topics discuss those five, and some subdivisions among them, in general terms. "Requesting SVC 99 Functions" presents the individual text unit keys you use to request SVC 99 functions, grouped within the general divisions discussed here.

## Dynamic Allocation

You can request one of two types of dynamic allocation: allocation by dsname or allocation by ddname.

### Dsname Allocation

***Dynamic allocation by dsname*** is equivalent to data set allocation during job step initiation; the SVC 99 parameter list is equivalent to a DD statement. In the parameter list, you request the allocation-by-dsname function by specifying verb code 01. You can request most of the JCL services that you can code in a DD statement – such as data set disposition, volume label information, expiration date, and SYSOUT destination – by specifying different text units in the parameter list. Figure 57 lists JCL DD statement facilities that *cannot* be used in dynamic allocation.

In addition, you can specify the following, which do not have a JCL equivalent, via SVC 99 text units:

- The password for a password-protected data set. If you specify the password in your program via SVC 99, the system need not prompt the operator.
- The permanently allocated attribute.
- The convertible attribute.
- Return of certain information.

| | |
|---|---|
| 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**<br>DCB — reference to ddname of a previous step<br>CYLOFL<br>NTM<br>RKP<br>DISP — PASS specification<br>DSN — reference to ddname (as in *.ddname)<br>ISAM area name<br>SPACE — ABSTR specification<br>UNIT — AFF<br>VOLUME — RETAIN specification<br>REF=ddname |

Figure 57. JCL DD Statement Facilities not Supported by Dynamic Allocation

Consult the detailed description of each text unit key (see "Requesting SVC 99 Functions") for the capabilities supported by the key. While a subparameter may be supported (for example, DCB=DSORG), all values of that subparameter may not be supported (for example, DCB=DSORG=IS).

See "Requesting SVC 99 Functions" for programming considerations for dynamic allocation by dsname.

## 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 processing sets the in-use bit on.

This type of dynamic allocation request is useful in time-sharing command processors for re-allocating groups of data sets that were allocated and concatenated by an earlier command processor but whose in-use bits were then turned off by the TMP.

In MVS, the HELP command processor uses ddname allocation to allocate the SYSHELP data set. As a user, you may allocate, in your logon procedure, a group of data sets to be searched for HELP information. These data sets are concatenated with the ddname of SYSHELP. When the TMP receives control after the LOGON command processor completes execution, the in-use bits of all the SYSHELP data sets are off. Then, when you issue the HELP command, the HELP command processor invokes the allocation-by-ddname function, specifying SYSHELP as the ddname. As a result, all of the data sets concatenated to SYSHELP will have the in-use bit turned on. (If the system cannot locate the SYSHELP ddname, the HELP command processor uses the allocation-by-dsname function to allocate the SYS1.HELP data set; it will be associated with a system-generated ddname.)

You request dynamic allocation by ddname by specifying verb code 06 and putting the ddname to be allocated in the SVC 99 parameter list. You are, in effect, asking the SVC 99 routines to use a specific existing allocation to satisfy your request.

In order for the SVC 99 routines 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 "The Permanently Concatenated Attribute" in the topic "Dynamic Concatenation" 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 request is failed and an error reason code is returned in the SVC 99 parameter list.

If the existing allocation meets the requirements, its in-use bit is turned on 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.

You may specify that an indication is to be returned if the existing allocation that satisfies the request is associated with a dummy data set.

## Dynamic Unallocation

The SVC 99 dynamic unallocation routines deallocate resources when you request it via the appropriate verb code. Two functions are available through dynamic unallocation:

- **Releasing a data set**, specified by verb code 02 with key 7 in the SVC 99 parameter list.

  Releasing a data set involves the following processes:

  - Disassociating the ddname from the data set name, which allows the ddname to be used in subsequent dynamic allocations.
  - Processing the data set disposition.
  - Releasing the data set for use by other jobs except when:
    - A temporary VIO data set is released at the end of the last step in which it is referenced.
    - A temporary non-VIO data set is released at the end of the job.
  - Freeing the unit(s) to which the data set was allocated.
  - Releasing the volume(s) on which the data set was allocated.

- **Removing the in-use attribute**, specified by verb code 02 with text unit key 8 in the SVC 99 parameter list.

  This function turns off the data set's in-use bit. When this processing is completed, the data set is referred to as "not-in-use." You can also request remove in-use processing based on task-id by specifying verb code 05; see "Removing the In-Use Attribute by Task-ID."

If you code verb code 02 without specifying key 7 or key 8, the dynamic unallocation routines *remove the in-use attribute* from data sets allocated through JCL, the time-sharing ALLOCATE command, or dynamically with the permanently allocated option; the routines *release* 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 will be satisfied in all but one case: the routines will not remove the in-use attribute from a non-permanently allocated, non-&dsname data set with a disposition of DELETE. Such a resource cannot be used to satisfy a subsequent request, so it will be released.

You can request either dynamic unallocation function for a dsname or a ddname.

**Dynamic Unallocation Processing:** The following considerations apply to all dynamic unallocation requests:

- If a data set is open, is a member of an open concatenated group, or is a private catalog, it will not be deallocated.
- When a SYSOUT data set is released, it is immediately made available for output (unless you specify an overriding disposition of DELETE, in which case the data set is deleted).

The following considerations apply to unallocation requests specifying a *dsname*:

- If no ddname is specified and the dsname is associated with more than one ddname, all associated data sets 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 a member name is specified with the dsname, only those associations containing both the member name and dsname are deallocated. (Both the member name and dsname text unit keys must be coded for a valid request.)

The following considerations apply to unallocation 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 or dsname and membername 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, 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. (An exception occurs when the concatenated group was generated by the system, as, for example, VSAM data sets spanning device types and GDG ALL groups. In these cases, the group is treated as a single resource.)

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.)

If a concatenated group has the permanently concatenated attribute and you specify a ddname with a dsname, a VSAM dsname, or GDG ALL, the entire group is released. If you specify a dsname with a VSAM dsname or GDG ALL, the request for dynamic unallocation fails.

**Changing Parameters at Dynamic Unallocation:** You can include with your dynamic unallocation request text units to change a data set'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 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
- HOLD/NOHOLD parameters
- Remote work station destination
- Disposition

Allocation disposition cannot be overridden for passed data sets, VSAM data sets, or system-named data sets. For all other types of data sets, the disposition specified on an unallocation request overrides the disposition specified at allocation.

If you request that a data set be cataloged when it is deallocated, you must provide linkage to the catalog data set. The system will not dynamically allocate the catalog data set. You can provide the necessary linkage by pre-allocating the catalog data set or by allocating it through a STEPCAT DD statement.

Members of partitioned data sets cannot be deleted with a disposition of DELETE; the entire data set is deleted. An overriding disposition of DELETE for data sets allocated as shared is invalid; the overriding disposition request is failed.

### 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, via 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. This function is used by, for example, the time-sharing terminal monitor program (TMP) to turn off the in-use bits of any data sets allocated by a command processor when the command processor completes execution.

## Dynamic Concatenation

Dynamic concatenation logically connects allocated data sets into a concatenated group. You request this function by specifying verb code 03 in the SVC 99 parameter list. You can only identify data sets to be concatenated 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 SVC 99 routines will concatenate 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 already associated with a concatenated group, that entire group will be 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 Permanently Concatenated Attribute

You can request that a concatenated group created via SVC 99 be assigned the permanently concatenated attribute. A concatenated group defined via JCL is automatically assigned the permanently concatenated attribute, as is a concatenated group defined by the system via JCL or SVC 99. A GDG ALL request and a request for a VSAM data set that spans device types are examples of the latter situation.

A group with the permanently concatenated attribute has the following characteristics:

- The group cannot be dynamically deconcatenated into its member data sets.
- 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.

## Dynamic Deconcatenation

Dynamic deconcatenation logically disconnects the members of a concatenated group. You request dynamic deconcatenation by specifying verb code 04 in the SVC 99 parameter list. You identify the concatenated group to be deconcatenated by specifying the ddname of the group.

The request for dynamic deconcatenation fails if the concatenated group is open. A permanently concatenated group, or members of a concatenated group that are permanently concatenated, remain concatenated.

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.

Dynamic deconcatenation has no effect on the in-use attributes associated with the members of the group.

## Dynamic Information Retrieval

Dynamic information retrieval provides you with information about your current allocation environment. You request this function by specifying verb code 07 in the SVC 99 parameter list. You can request information about ddnames or dsnames.

In addition, you can ask for information about any or all of your currently-allocated requests by specifying a relative request number. For example, you could 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.

You can request the following kinds of information using SVC 99 verb code 07:

- Data set name
- Ddname
- Member name
- Data set organization
- Status
- Normal disposition
- Conditional disposition
- Attribute status, including the permanently allocated, in-use, permanently concatenated, convertible, and dynamically allocated attributes
- Data set types, including dummy, SYSIN, SYSOUT, and allocation of the user's terminal as an I/O device
- The number of resources held in anticipation of reuse that exceeds the control value; that is, the number of existing allocations that must be deallocated before a request for a new allocation can be satisfied
- Whether or not the allocation is the last relative request.

# Installation Options For SVC 99 Functions

This section describes the values and options your installation might want to modify in order to control SVC 99 processing. The values and options discussed in the following topics are:

- Default values for space and unit information
- Default values for the TIOT
- Mounting volumes and bringing devices online
- Installation validation routines.

## Space and Unit Defaults

This section describes how to change the allocation default values for space and unit. It also describes remote user workstation and TIOT defaulting.

**Existing Default Values for Space:** If no space information is specified on a request for a new direct access data set, and the request is eligible to MSS exclusively (with MSVGP specified), the SVC 99 routines use the MSVGP defaults.

If the request is not eligible to MSS exclusively, or MSVGP is not specified, the defaults are:

- Block length of 1000
- 10 primary blocks and 50 secondary blocks
- Release unused space (RLSE) specification

These space defaults are contained in the allocation default CSECT IEFAB445 (load module IEFAB445). The contents of the module are as follows:

**Note:** Beginning at offset zero; the hexadecimal numbers in parentheses show the values supplied by IBM.

- Three bytes for the primary quantity value ('00000A')
- Three bytes for the secondary quantity value ('000032')
- Three bytes for the average block length ('0003E8')
- Three bytes for the number of directory blocks ('000000')
- One byte of flags with the following bit meanings:
  - bit 0: TRK (0)
  - bit 1: CYL (0)
  - bit 2: blocklength (1)
  - bit 3: RLSE (1)
  - bit 4: CONTIG (0)
  - bit 5: MXIG (0)
  - bit 6: ALX (0)
  - bit 7: ROUND (0)

**Existing Default Values for Unit:** If a time-sharing user's dynamic allocation request does not include unit information, the SVC 99 routines obtain a unit description from the UADS entry.

If the user is not a time-sharing user, or if the UADS entry does not contain a unit description, 'SYSALLDA' (the system's esoteric name indicating all direct access devices) is the default. This default is contained in the allocation default CSECT IEFAB445, in the eight bytes beginning at offset 13 (decimal).

The unit description you supply in your dynamic allocation request can override the unit type for a cataloged data set. The unit description from the UADS, however, cannot override the unit information in the catalog.

**Default Value for a Remote User Work Station:** If a remote user work station – or destination – is not specified by a time-sharing user allocating a SYSOUT data set, the SVC 99 routines obtain a default value from the UADS entry.

**Default Value for The TIOT:** The size of the TIOT is determined by the DEFTIOTS field in the allocation default CSECT (IEFAB445). The IBM-supplied default for the size of the TIOT is 32K. An installation may change that to a larger (maximum 64K) value or a smaller (minimum 16K) value. This default is contained in the allocation default CSECT IEFAB445, in the byte beginning at offset 23 (decimal).

The size of the TIOT controls how many DDs are allowed per jobstep. By specifying any integer from 16 to 64 as the value of the DEFTIOTS field, the user controls the DD allowance. The following table shows the relationship between the size of the TIOT and the maximum number of DDs allowed:

| DEFTIOTS Dec | (Hex) | Size of TIOT | Maximum number of DDs allowed |
|---|---|---|---|
| 16 | 10 | 16384 (16K) | 816 |
| 24 | 18 | 24576 (24K) | 1225 |
| 32 | 20 | 32768 (32K) | 1635 |
| 40 | 28 | 40960 (40K) | 2045 |
| 48 | 30 | 49152 (48K) | 2454 |
| 56 | 38 | 57344 (56K) | 2864 |
| 64 | 40 | 65536 (64K) | 3273 |

## Mounting Volumes and Bringing Devices Online

Dynamic allocation processing can bring devices online and have volumes mounted.

This function is optional for time-sharing 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 via the UADS entries.

Other users of SVC 99, however, can always have volumes mounted and devices brought online. If you do not want this function performed during dynamic allocation, you can indicate in the SVC 99 parameter list that volumes are not to be mounted and that devices are not to be brought online for a request.

In addition, the operator may inform the dynamic allocation routines that a volume is not to be mounted or that a device is not to be brought online. If the operator thus prevents the mounting or bringing online of a volume or device, the allocation request fails.

If you allow volume mounting, the SVC 99 routines will wait for tape volumes to be mounted. Common allocation processing, by contrast, does not wait for tape volumes to be mounted. When a volume is mounted for a dynamic allocation request, OPEN verifies that it is the correct volume.

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 SVC 99

An exit (IEFDB401) from the allocation control routine provides for a user-written routine to validate or alter any request to SVC 99. The routine is entered for all system and user SVC 99 requests. You must code it so it does not interfere with system requests.

Your validation routine can test and modify the SVC 99 input request, and it can indicate through a return code whether processing of the request is to continue. For example, the routine might perform the following functions:

- Control the amount of direct access space requested
- Check for authorization to use specified units
- Check for authorization to use certain data sets
- Check for authorization to hold certain resources for reuse

See *SPL: User Exits* for information about programming conventions that the input validation routine must observe.

# Requesting SVC 99 Functions

To request an SVC 99 function, you must code the DYNALLOC macro (it has no operands) and supply the SVC 99 parameter list. The SVC 99 routines perform requested functions based on the information you provide in the parameter list, in the form of text unit keys.

This topic consists primarily of descriptions of the various text unit keys you use to request SVC 99 functions. It also includes a discussion of the parameter list itself, a detailed review of the process of dynamic allocation by dsname, the SVC 99 return codes, and some general considerations for using SVC 99. An example of a dynamic allocation request is provided at the end of this topic.

## Programming Considerations When Using SVC 99

Before deciding to use any of the SVC 99 functions, you should consider the environment of the program that invokes SVC 99. Your program interacts with the job entry subsystem, with the MVS initiator, with data management functions, and with the common allocation functions of the MVS scheduler, in addition to the SVC 99 routines themselves. If the program is a system routine, there are cross-memory considerations, as well.

You must make sure your routine does not interfere with the normal functioning of the operating system. Conversely, you should be aware that operating system functions could affect the interface between your program and the SVC 99 routines; and that the SVC 99 routines could affect the functioning of your program.

Following are some of the things you need to consider when preparing to use SVC 99. Other considerations are included with the topics to which they apply.

- **Serialization of Resources**

  Your program might serialize the same resources as SVC 99. The SVC 99 routines can serialize the following resources, depending on the path taken in SVC 99 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 SVC 99**

  System routines invoked by various paths of SVC 99 processing might also serialize a system resource. Some of the system functions invoked by SVC 99 processing are LOCATE, OBTAIN, CATALOG, SCRATCH, DADSM Allocate, and MSS Interface (SVC 126).

- **No Cross Memory Support for SVC 99**

  SVC 99 does not work in cross memory mode.

- **Enqueuing on the SYSZTIOT; Avoiding 138 Abends**

  Avoid requesting SVC 99 functions in routines that run under the control of an interruption request block (IRB), especially when the program that might be interrupted issues OPEN, OPENJ, CLOSE, EOV, or FEOV, or any other SVC that enqueues on the SYSZTIOT. An SVC 99 request issued in such an environment can cause a 138 abend when SVC 99 tries to enqueue on the SYSZTIOT resource.

  For the same reason, user exits for OPEN/CLOSE/EOV, or for any other routines that enqueue on SYSZTIOT, should not issue SVC 99 requests.

- **Avoiding 0B0 Abends**

  Programs that issue SVC 99 should not receive control during START (initialization) processing for LOGONs, MOUNTs, or started tasks.

  Programs that get control during START processing (user 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 0B0 abend.

  Subsystems that receive control during step allocation as a result of the SUBSYS parameter should not issue SVC 99; to do so might cause an 0B0 abend.

- Accessing ICF CATALOGS, CVOLs and VSAM Private Catalogs

  Programs that get control during 'START' (Initialization, User 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 MSGIEC331I, RC4, RC84.

- **SMS Consideration**

  Programs that issue SVC 99 in an environment with the storage management subsystem (SMS) active should request the message processing function of SVC 99 in order to obtain error messages relating to a request.

- **JES Consideration**

  The program that requests SVC 99 functions must not receive control when the job entry subsystem is being started; an unending wait could result, causing the system to crash.

- **Effect of Outstanding STIMER or STIMERM**

  The program that requests SVC 99 functions must not have an STIMER macro outstanding, because certain paths in SVC 99 issue another STIMER, which causes an overlay of the first STIMER. This situation causes the program that issued the SVC 99 request never to receive control because of the expiration of the timer.

  One STIMER ID must be available for allocation processing. If an ID is unavailable, an X'05C' abend with a return code of 6 may result.

- **Considerations for System Routines**

  System routines that invoke SVC 99 functions should be aware that a non-zero return code might be returned. Because system routines cannot always diagnose non-zero return codes, the system routine should print an error message including the error code (S99ERROR) and information code (S99INFO) fields from the SVC 99 request block (S99RB).

- **Accessing CVOLs or VSAM Private Catalogs**

  Routines should not allocate data sets that are cataloged in OS CVOLs or VSAM private catalogs to long-running tasks, because the private catalog or CVOL will be allocated and will remain allocated until the step terminates. This is especially important in installation exits for system tasks, because the private catalog or CVOL might be allocated to an initiator or subsystem such as JES2 or JES3.

Volumes that contain a CVOL or VSAM private catalog for data sets allocated to long-running steps should be assigned the permanently resident attribute.

- **Changes to the TIOT By SVC 99 Routines**

  SVC 99 routines might cause changes to the task input/output table (TIOT). Depending on the function requested via SVC 99, an entry could be added, deleted, or reordered; you cannot assume a fixed order for TIOT entries.

  You should make sure a problem program is aware of changes to the TIOT, especially if the EXTRACT macro is also being used within that program, or in the program that will gain control when SVC 99 processing is finished.

  If you need to reference TIOT entries after SVC 99 is invoked, do so via the data set association block (DSAB) chain, which is pointed to by the DSAB queue descriptor block (DSAB QDB) field, JSCDSABQ, in the active JSCB.

  Datasets that are cataloged in a private catalog or CVOL should not be dynamically allocated in the JES3 address space. If they are dynamically allocated, a system deadlock may occur involving SYSZTIOT, which is required for the SVC 99, and SYSZPCCB, which is required in the modification of the private catalog control block (PCCB).

## SVC 99 Parameter List

When you code DYNALLOC, you must supply a parameter list. The SVC 99 parameter list includes a request block, request block extension, text pointers, and text units. The request block, text points, and text units are required. The request block extension is required only if you need to use the message processing function of SVC 99.

The request block indicates the function you want the SVC 99 routines to perform, and the extension contains message processing information. The text pointers contain the addresses of the text unit keys, and the text units contain the keys and parameters for the SVC 99 functions. Figure 58 illustrates the structure of the SVC 99 parameter list.

IBM supplies two macros, IEFZB4D0 and IEFZB4D2, to aid in constructing the SVC 99 parameter list. IEFZB4D0 provides symbolic names (DSECTs) for the positional information in the structure; IEFZB4D2 provides mnemonics for the text unit keyword values. The names in Figure 58 are those assigned by the macro IEFZB4D0.

On entry to SVC 99, register 1 must point to a pointer to the request block.

The pointers to the parameter list, the request block, the extension, and the text units must be created in storage with the same key as the one for the caller of SVC 99.

In addition, the request block, the extension, and any information retrieval text units must be in non-store-protected storage. This is to prevent an 0C4 abend when SVC 99 copies the caller's parameters into its own work area and then restores the information retrieval text units into the caller's storage.

**Note:** SVC 99 only restores information retrieval text units originally specified by the caller and validated by the IEFDB401 exit routine (see *SPL: User Exits*). Text units added or modified by IEFDB401 are not copied back to the calling program's storage.

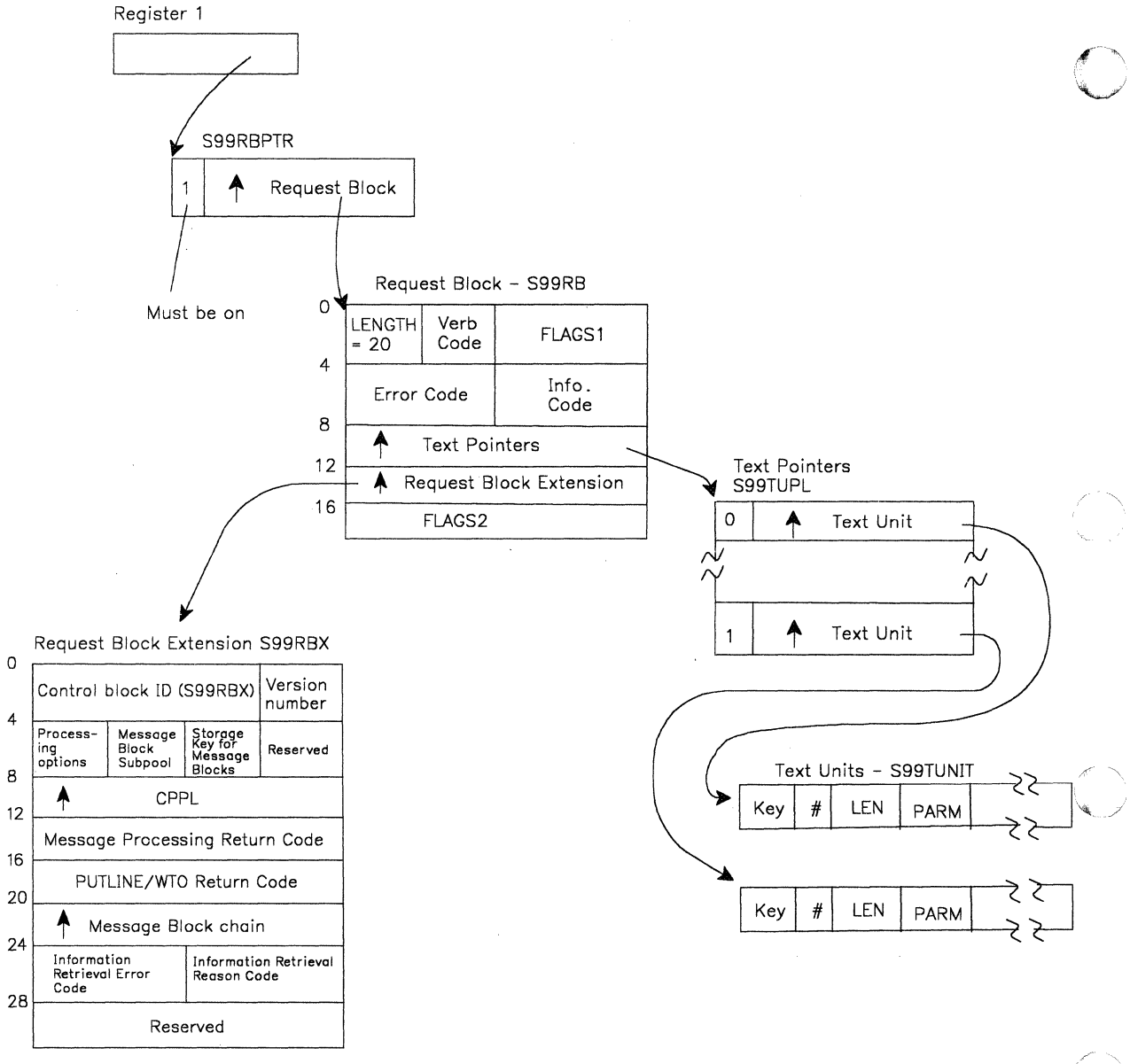



Figure 58. Structure of the SVC 99 Parameter List

## Request Block Pointer

The request block pointer, DSECT name S99RBP, is a single fullword containing the address of the SVC 99 request block. IEFZB4D0 assigns the label S99RBPTR to the address. The high-order bit in this field must be set to one.

## Request Block

The request block 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):

- **LENGTH (S99RBLN)** A one-byte field containing the length of the request block. The length is always 20 bytes.
- **VERB CODE (S99VERB)** A one-byte field that identifies the SVC 99 function to be performed. You may specify 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 |

- **FLAGS1 (S99FLAG1)** A two-byte field that instructs the system on how to satisfy dsname allocation requests. The meaning of the bits in the field are as follows:

| 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 offline devices. (This bit overrides S99MOUNT and S99OFFLN in FLAGS2.) If this bit is one and the request causes a private catalog to be allocated, mounting will not be allowed for that catalog. |
| 3 | S99JBSYS | Treat the data set as part of the job's normal output. The data set is not expected to be dynamically deallocated (spun off). This flag is used for SYSOUT data sets. 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. |
| 5-16 | | Reserved; must be zero |

**Note:** The FLAGS1 indicators (except S99CNENQ) are used only for dsname allocation requests.

- **ERROR CODE (S99ERROR)** A two-byte field that SVC 99 uses to return error reason codes. See "SVC 99 Return Codes."
- **INFO CODE (S99INFO)** This two-byte field is used by SVC 99 to return information reason codes. See "SVC 99 Return Codes."
- **TEXT POINTERS ADDRESS (S99TXTPP)** A fullword field containing the address of a list of pointers to the text units.
- **REQUEST BLOCK EXTENSION ADDRESS (S99RBX)** A fullword field containing the address of the request block extension.
- **FLAGS2 (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 system storage protection key (0-7).
  - It must be in supervisor state.
  - It must be APF-authorized.

The meanings of the FLAGS2 bits are:

| Bit | Bit Name | Meaning When On |
|---|---|---|
| 0 | S99WTVOL | Wait for volumes. |
| 1 | S99WTDSN | Wait for dsname. (See Note 2.) |
| 2 | S99NORES | Do not reserve data sets. |
| 3 | S99WTUNT | Wait for units. (See note 3.) |
| 4* | S99OFFLN | Consider offline devices. The system ignores this bit if S99NOMNT in FLAGS1 is on. For a batch user, or if a time-sharing user has the mount attribute in his UADS entry, the system proceeds as if this bit were on, regardless of the setting. |
| 5 | S99TIONQ | TIOT ENQ already performed. |
| 6 | S99CATLG | Set special catalog data set indicators. |
| 7* | S99MOUNT | Volumes may be mounted. The system ignores this bit if S99NOMNT in FLAGS1 is on. For a batch user, or if a time-sharing user has the mount attribute in her UADS entry, the system proceeds as if this bit were on, regardless of the setting. |
| 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 | S99PCINT | Allocate a private catalog on behalf of the initiator. |
| 10-31 | | Reserved. Must be zero. |

*These fields override the NOMOUNT option from the TSO user attribute data set (UADS).

**Notes:**

1. The FLAGS2 indicators (except S99TIONQ) are used only for dsname allocation requests.
2. In a JES3 environment, authorizing a dynamic allocation request to wait for data set availability might cause a system interlock.
3. Authorizing a dynamic allocation request to wait for currently allocated devices may cause the job to hang if the only devices available are already allocated to this job.

## Request Block Extension

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):

- **Request Block Extension Identifier (S99EID)** A six-byte field containing the request block extension identifier, which can be obtained from the S99RBXID field in IEFZB8D0.
- **Version (S99EVER)** A one-byte version number of the request block extension, which can be obtained from the S99RBXVR field in IEFZB4D0.

- **Processing Options (S99EOPTS)** A one-byte field that defines the SVC 99 message processing options. Select the processing options that you want by setting bits in this field as follows:

| Bit | Bit name | Meaning |
|---|---|---|
| 0 | S99EIMSG | The system issues error messages **before** control returns to the caller of SVC 99. |
| 1 | S99ERMSG | The system returns the messages to the caller of SVC 99, but the system does not issue the messages unless S99EIMSG is set. |
| 2 | S99ELSTO | The system returns the message blocks to the caller in the first 16 megabytes of storage. |
| 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 | The system uses a WTO macro instruction to issue the error messages if this bit is set. Otherwise, the system uses a TSO PUTLINE command to issue the messages. |
| 6-7 | | Reserved. These bits must be zero. |

- **Message Block Subpool (S99ESUBP)** A one-byte field that defines the storage subpool containing the message blocks returned to the caller. This field is ignored unless the processing option S99EMSUB is indicated. If the caller does 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 hexadecimal.
- **Storage Key (S99EKEY)** A one-byte storage key for the storage in which the message blocks are returned. This field is ignored unless the processing option S99EMKEY is specified. If the caller does 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 hexadecimal.
- **Severity Level (S99EMGSV)** A one-byte field that defines the minimum severity of the messages that should be processed by SVC 99. The severity levels, which are informational, warning, and severe, are defined by S99XINFO, S99XWARN, and S99XSEVE in the IEFZB4D0.
- **Number of message blocks returned (S99ENMSG)** A one-byte field containing the number of message blocks returned from SVC99.
- **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.
- **Reserved. (S99ERCR)** A one-byte reserved field containing zero.
- **Reserved. (S99ERCM)** A one-byte reserved field containing zero.

- **Message processing reason code (S99ERCO)** A one-byte reason code that explains the failure of a message processing function.

| Hexadecimal code | Meaning |
|---|---|
| 03 | WTO failed. |
| 04 | PUTLINE failed. |
| 05 | Unable to obtain storage for message blocks. |
| 06 | Unable to obtain storage for PUTLINE macro. |
| 07 | A CPPL address was not supplied for the PUTLINE message output function. |
| 08 | The message block chain was invalid. |
| 09 | Message extraction failed because the message block chain was invalid. |

- **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 |
|---|---|
| 01 | Storage cannot be freed because FREEMAIN failed. |
| 02 | Storage cannot be freed because of an invalid message block chain. |

- **PUTLINE/WTO macro return code (S99EWRC)** The fullword return code from the WTO or PUTLINE macro, which are the macros that issue the messages.
- **Message Block Chain Address (S99EMSGP)** A fullword that contains the address of a chain of message blocks.
- **Information Retrieval Error Code. (S99EERR)** A two-byte code that explains errors found in information retrieval text units of dynamic allocation. This two-byte code applies only to verb code 01. See the SVC 99 return codes on page 1-259.
- **Information Retrieval Information Code (S99EINFO)** A two-byte field containing an erroneous text unit. The text unit is an information retrieval text unit. This two-byte code applies only to verb code 01. See the SVC 99 return codes on page 1-259.
- **Reserved. (S99ERSV2)** A fullword that contains zero.

## Text Pointers

The text pointer part of the parameter list is a variable-length list of fullwords containing pointers to the text units. You indicate the end of the list by setting the high-order bit of the last pointer to one. A fullword of zeros is ignored.

Mapping macro IEFZB4D0 assigns the DSECT name S99TUPL to the list, the label S99TUPTR to each pointer in the list, and label S99TUPLN to an equate that allows you to turn on the end-of-list indicator.

## Text Units

Each SVC 99 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 containing a unique binary number that identifies the type of information to be found in the PARM subfield. For example, a key of '0004' for a dsname allocation request indicates that the value of the PARM subfield specifies data set status. SVC 99 ignores a KEY field of zero. See "Text Units by Function" for a description of the text units that can be coded for each SVC 99 function.
- **NUMBER (S99TUNUM)** A two-byte binary number specifying the number of length and parameter combinations in the text unit. If a key of zero is specified, S99TUNUM must also be zero.
- **COMBINATION (S99UENT)** 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 binary number specifying the length of the following parameter field.
- **PARM (S99TUPAR)** A variable-length field in which you put the parameter information identified by the value in the KEY field. See "Text Units by Function" for a description of the values you can code for each text unit key.

The following notes apply to the structure of the text units; you will find rules for coding specific text units in "Text Units by Function."

**Notes:**

1. Special characters – of the type requiring apostrophes in JCL statements – are not valid in PARM values, except in the DALUSRID text unit.
2. Parameters whose values consist of alphameric and national characters may include trailing blanks.
3. The text units may be in any order.
4. Each function of SVC 99 has an associated set of text units, and each set is independent of any other. For example, the functions of allocation and unallocation may both use a KEY value of '0007', but that value does not necessarily have the same meaning for both functions.

# Detailed Review of Dsname Allocation Processing

The major function performed by the SVC 99 routines, and the function most often requested of them, is that of dynamically allocating a data set/resource according to its data set name (dsname). Following is a detailed discussion of the processing the SVC 99 routines perform in satisfying dsname allocation requests.

When you invoke SVC 99 to perform dsname dynamic allocation, an "allocation environment" already exists for your request. It consists of the allocation requests made via your JCL or internal dynamic allocation, that have not yet been deallocated. The system considers these resources to be **existing allocations**, and goes to them first to fill your SVC 99 requests.

For dsname allocation, the SVC 99 routines first check for environmental conflicts by noting the types of resources that are currently available to the task that includes your program. The routines then try to satisfy the request with an existing allocation that matches or can be made to match the request. (See the discussion of the convertible attribute in the "Dynamic Allocation" topic.) If the routines cannot make the match, they proceed with a new allocation. If an existing allocation can be used, much allocation processing is avoided.

## Checking for Environmental Conflicts

The SVC 99 routines cannot satisfy a dsname allocation request that is in conflict with your existing allocation environment. Following is a list of the environmental conflicts that can cause your request to fail:

- The specified ddname is associated with an existing allocation that is in use.
- The specified ddname is associated with one of a group of concatenated data sets defined as permanently concatenated. (For a definition of permanently concatenated, see "The Permanently Concatenated Attribute.")
- The request specifies 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.)
- A status of OLD or SHR is specified for a dsname associated with an existing allocation that is not permanently allocated, not in-use, and has a disposition of DELETE. (This is not a conflict if the request specifies a different volume serial number.)
- The specified ddname 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."

## Using an Existing Allocation

If possible, the SVC 99 routines 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.

In order to be satisfied by an existing allocation, your request must be one of the following:

- A request for an explicit data set name (dsname), or
- A request for the allocation of your terminal as an I/O device, or
- A request for a dummy data set

In order 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
- 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

  **Note:** MSVGP is ignored if an existing allocation is used.

In order 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 either have the convertible attribute 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 a status of MOD is specified in the request, 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," only the first item need be true.

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 the SVC 99 routines select to satisfy your request.

If you did not specify a ddname, the SVC 99 routines select the matching existing allocation whose in-use bit was most recently turned off. (Data sets allocated via JCL are considered to have had their in-use attributes removed at step allocation.)

It could happen that an existing allocation does not match your request even though it is associated with the same ddname you specify. Since the ddname is going to be associated with the resource that is allocated to your program, the system gives the rejected 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 in the following cases:

- The existing allocation is in use.
- The existing allocation is open.
- The existing allocation does not have the convertible attribute.

- The existing allocation 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 the dynamic allocation routines use an existing allocation to satisfy a dsname allocation request, some of the parameters of the existing allocation might have to be changed to reflect the parameters specified in the request. Only existing allocations that were dynamically allocated, with the convertible attribute, can have their parameters changed. Resources allocated via JCL or the TSO ALLOCATE command cannot have their parameters changed (with the exception of status and disposition specified via JCL), but they may be used if no changes are necessary.

Note that, if your request does not specify the permanently allocated attribute, the allocated resource is automatically assigned the convertible attribute. An allocation request may specify both the permanently allocated and the convertible attributes in its parameter list.

The following parameters are eligible for change by the SVC 99 routines:

- Ddname
- Member name
- Status
- Normal disposition
- Conditional disposition
- Space
- Unallocation at CLOSE
- Input only
- Output only
- DCB attributes
- Password
- Permanently allocated attribute

No other parameters may be changed.

**Notes:**

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, or the next job enqueued on the data set has requested exclusive use of the data set.
2. You cannot change the parameters on an explicitly-referenced OUTPUT JCL statement (DALOUTPT).

## Using a New Allocation

The dynamic allocation routines attempt a **new allocation** when they cannot satisfy your request with an existing allocation. New allocations cannot be processed by the dynamic allocation routines while a job step holds (for possible reuse) more dynamically allocated resources than permitted. The number of allocated resources permitted is determined in two ways:

- The maximum number of concurrent allocation requests allowed per job step or time-sharing terminal session, including dynamic requests and requests made via JCL, is equal to the maximum number of single-unit DD statements allowed by the TIOT size specified at IPL. For information on how the TIOT size is related to the maximum number of DD statements, see "Installation Options For SVC 99 Functions" on page 1-237.
- The control limit set by the DYNAMNBR parameter on the JCL EXEC statement plus the number of DD statements limits the number of resources that can be held for reuse (that is, resources that are allocated but whose in-use bits are off).

When the maximum number of resources have been allocated and you request additional allocations, the dynamic allocation routines automatically attempt to deallocate enough resources to meet the control limit.

### Automatic Unallocation of Resources Held for Re-use

The only resources eligible for automatic deallocation are those that were allocated dynamically without the permanently allocated attribute and whose in-use bit has been turned off. (Resources allocated through JCL and through the time-sharing ALLOCATE command are not eligible because they automatically have the permanently allocated attribute.)

When many resources are eligible for automatic deallocation, the dynamic allocation routines choose those that have been designated as not-in-use for the longest time. These are deallocated and the new allocation is processed.

If the control value 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 performed.

## Considerations When Requesting Dsname Allocation

- If you do not specify a ddname, the system generates one. The ddname created consists of the characters 'SYS' followed by five digits.
- You may specify passwords as part of a dynamic allocation request to bypass prompting the operator.
- 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.
- 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.
- Rather than wait for another user to release a data set, volume, or device in order to obtain use of it, the dynamic allocation routines fail a request by an unauthorized program. If an authorized program specifically requests a wait, the routines will wait.
- You can request that the ddname, data set name, and volume serial number assigned by the allocation routines be returned in the SVC 99 parameter list.
- You can also request that the data set organization (DSORG) of the allocated data set be returned in the SVC 99 parameter list. The SVC 99 routines return whatever you specify as the DSORG, if any. If you do not specify a DSORG on the allocation request, the system assigns and returns data set organizations 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 directory space quantity; otherwise, the data set is assigned the DSORG of 'PS'.
  - 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 is failed.
  - For other types of allocation requests where you do not specify a DSORG for the data set, the system returns zeros in the SVC 99 parameter list field.

- For time-sharing users allocating new data sets using the TSO ALLOCATE command, DSORG is defaulted to partitioned organization (PO) if a directory quantity is specified, or to physical sequential (PS) otherwise.
- You cannot create ISAM data sets through dynamic allocation.
- You cannot create VSAM data sets through dynamic allocation unless the storage management subsystem is active.
- If you request an allocation by dsname and the dsname is already allocated but not available, the SVC 99 routines copy the unit and volume information from the existing allocation and associate the information with your request.
- An allocation of a GDG data set will refer to the same data set for the life of the job (or TSO logon session), even if another generation is added during the job.
- The dynamic allocation routines will not use passed data set information to retrieve volume information.

## 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 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.

This book 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 DAIRFAIL to convert the reason code into the message. DAIRFAIL is an IBM-supplied program that is described in *TSO/E Version 2 Programming Services*.

### 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 environment, the end user is a TSO 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. A message can be a single message or a composite of several messages. You can request dynamic allocation to:

- Send the message directly to the end user, or
- Return the message to your program

You can request to send the message to a TSO terminal or to the system job log. You can ask dynamic allocation not to send low-severity messages, such as informational messages that do not denote errors.

Some installations might need to write special message-sending programs. In this case, request to have the message returned to your program instead of being sent. Request this by setting a control field in the request block extension. When you request the message to be returned, you can control the severity level of the returned messages. After dynamic allocation gives control back to your program, use IEFDB476 to help process the message.

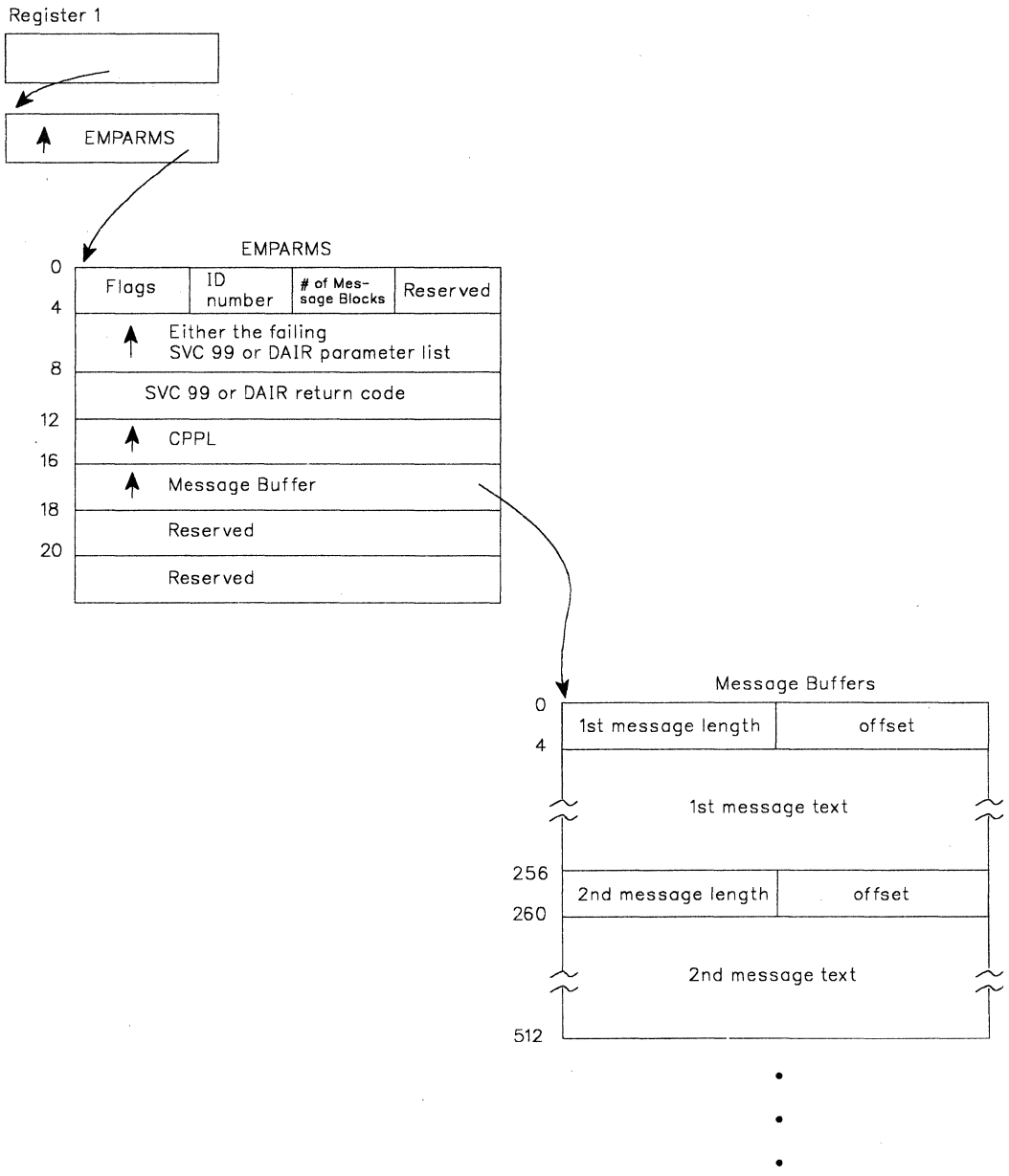
## Functions of the IEFDB476 Program

The three main functions of IEFDB476 are:

- **Extracting messages** — This function extracts the message from the system, formats it, and places it in a location that you specify. To extract, obtain an area of storage (GETMAIN macro) large enough to hold all the messages that you want to extract. When you invoke GETMAIN, note that the message might not be one but several separate messages. You must obtain 256 bytes for each separate message. The number of separate messages is in a request block extension field. When you invoke IEFDB476, you pass the location of the first 256-byte area. IEFDB476 places the extracted messages in contiguous 256-byte areas. Then your program can process the messages as required.
- **Sending Messages** — This function sends the message to the end user. When you use this function, you must identify the end user that receives the message.
- **Freeing Storage** — This function frees the storage area where the system keeps the message. To use this function, you indicate that the storage is *not* to be freed. Otherwise, the storage is freed by default.

  **Note:** *Besides extracting messages, sending messages, and freeing storage, IEFDB476 can also convert a return 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.*

To invoke IEFDB476, you must be in 31-bit addressing mode. General register 1 must contain the address of a four-byte parameter list that contains the address of the input parameter list:


Register 1
EMPARMS
EMPARMS
0
4
8
12
16
18
20
Flags
ID number
# of Mes-sage Blocks
Reserved
Either the failing SVC 99 or DAIR parameter list
SVC 99 or DAIR return code
CPPL
Message Buffer
Reserved
Reserved
Message Buffers
0
4
1st message length
offset
1st message text
256
260
2nd message length
offset
2nd message text
512

The input to IEFDB476 is defined by the EMDSECT1 dsect in the IEFZB476 macro:

- **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 via a PUTLINE |
| 1 | EMWTP | Issue the messages via a 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 SVC 99 request block extension. |
| 4-7 | EMRSV01 | Reserved |

- **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 an SVC 99 error |
| 51 | EMFREE | FREE command with an SVC 99 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 SVC 99 is in field S99ENMSG of the SVC 99 request block extension. The default count is 2 to maintain compatibility with DAIRFAIL.
- **Reserved** — A one-byte field containing zero.
- **Parameter list address (EMS99RBP)** — A four-byte field containing the address of either the failing SVC 99 parameter list or the failing DAIR parameter list.
- **Return code (EMRETCOD)** — A four-byte field containing the SVC 99 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.
- **Reserved** — An eight-byte field containing zero.

The message buffer array is defined by the EMDSECT3 dsect in the IEFZB476 macro:

- **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 251-byte field containing the returned message text.
- **Reserved**— A one-byte field containing zero.

**Note:** This dsect is provided for compatibility with DAIRFAIL.

The message buffers are defined by the EMDSECT2 dsect in the IEFZB476 macro:

- **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.
- **Reserved** — A one-byte field containing zero.
- **First message text (EMBUFT1)** — A 251-byte field containing the first returned message text.
- **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 zero.
- **Second message text (EMBUFT2)** — A 251-byte field containing the second returned message text.
- **Reserved** — A one-byte field containing zero.

The Dynamic Allocation Error Message Processor program (IEFDB476) produces return codes in general register 15. (It does not produce reason codes unless the SVC 99 caller uses a request block extension. The reason codes, which are in the reason code fields of the request block extension, are described on page 1-246.)

| Hexadecimal Return Code | Meaning |
|---|---|
| 00 | The request is successful |
| 04 | The identification number of the caller is invalid |
| 08 | An error occurred in PUTLINE or WTO while outputting a message. The PUTLINE or WTO return code, if any, is in the S99EWRC field of the SVC 99 request block extension. |
| 0C | The program (IEFDB476) is unable to return messages |
| 10 | The program (IEFDB476) is unable to free the storage associated with the message block chained out of the SVC 99 request block extension.<br><br>**Note:** hexadecimal return code 10 is applicable only to SVC 99 callers that have a request block extension. |

## SVC 99 Return Codes

**Note:** The data area labels used in this topic are assigned by macros IEFZB4D0 and IEFZB4D2.

When the SVC 99 routines return 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 Figure 59.

| Code | Meaning |
|---|---|
| 0 | Successful completion; there will also be an information reason code if a non-terminating error occurred during request processing. |
| 4 | An error resulted from the current environment, the unavailability of a system resource, or a system routine failure; there will also be an error reason code. |
| 8 | The installation validation routine denied this request. (See "Installation Input Validation Routine" for additional information.) |
| 12 | The error is due to an invalid parameter list; there will also be an error reason code from class 3. (Class 3 reason codes are listed in Figure 60.) |

Figure 59. SVC 99 Return Codes

The next two topics describe the information and error reason codes that the SVC 99 routines return in the SVC 99 request block. The last part of this chapter contains the descriptions of the SVC 99 text units, and an example of a dynamic allocation request.

### Information Reason Codes

When the SVC 99 routines encounter a non-terminating error during processing, an information reason code appears in the request block field labelled S99INFO. The possible codes and their meanings are:

| Code | Meaning |
|---|---|
| **0004** | Reserved |
| **0008** | Overriding disposition ignored for one of the following reasons:<br>• Data set was originally allocated with a disposition of PASS<br>• Data set is a non-subsystem data set that has a system-generated name; you cannot override disposition on this type of data set<br>• Data set is a VSAM data set and the storage management subsystem is not active.<br>In these cases, the data set is deallocated using the disposition specified when the request was allocated. |
| **000C-001C** | Reserved |

**002n** The data set was successfully deallocated but processing of the requested CATLG or UNCATLG disposition was unsuccessful. The digit "n" is a code representing the reason for the failure. The possible codes and their meanings are:

| Code | Meaning |
|---|---|
| 1 | A control volume is required; a utility program must be used to catalog the data set. |
| 2 | The data set to be cataloged is already cataloged; or the data set to be uncataloged could not be located; or no change was made to the volume serial list of a data set with a disposition of CATLG. |
| 3 | The specified index does not exist. |
| 4 | The data set could not be cataloged because the space was not available in the catalog. |
| 5 | Not enough storage was available to perform the specified cataloging. |
| 6 | The data set to be cataloged in a generation index is improperly named. |
| 7 | The data set to be cataloged has not been opened; no density information is available (for dual density tape requests only). |
| 8 | Reserved |
| 9 | An uncorrectable I/O error occurred in reading or writing the catalog. |

**003n** The data set was successfully deallocated but processing of the requested DELETE disposition was unsuccessful. The digit "n" is a code representing the reason for the failure. The possible codes and their meanings are:

| Code | Meaning |
|---|---|
| 1 | The expiration date has not occurred. |
| 2 | Reserved |
| 3 | Reserved |
| 4 | No device was available for mounting the volume during deletion. |
| 5 | Not enough storage was available to perform the specified deletion. |
| 6 | Either no volumes were mounted or volumes that were mounted could not be demounted to permit the remaining volumes to be mounted. |
| 8 | The SCRATCH routine returned an error code. If the user's JOB statement requested allocation/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. |

## Error Reason Codes

When the SVC 99 routines return a nonzero return code in register 15, the request block field labelled S99ERROR contains a code that explains the reason for the return code.

Error reason codes are divided into the following classes:

| Class | Description |
|---|---|
| 1 | Reserved |
| 2 | Unavailable system resource |
| 3 | Invalid parameter list |
| 4 | Environmental error |
| 5 | Reserved |
| 6 | Reserved |
| 7 | System routine error |

The error reason codes are shown in Figure 60 through Figure 63. 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.

**CLASS 2 CODES (UNAVAILABLE SYSTEM RESOURCE)**

| Hex Code | (Decimal) | Meaning |
|---|---|---|
| 0204 | (516) | Real storage unavailable. (dsname allocation) |
| 0208 | (520) | Reserved. |
| 020C | (524) | Request for exclusive use of a shared data set cannot be honored. (dsname allocation)[1] |
| 0210 | (528) | Requested data set unavailable. The data set is allocated to another job and its usage attribute conflicts with this request. (dsname allocation)[1] |
| 0214 | (532) | Unit(s) not available; or, if allocating an internal reader, all defined internal readers are already allocated. (dsname allocation)[1] |
| 0218 | (536) | Specified volume or an acceptable volume is not mounted, and user does not have volume mounting authorization through SVC 99 request.(dsname allocation)[1] |
| 021C | (540) | Unit name specified is undefined. (dsname allocation) |
| 0220 | (544) | Requested volume not available. (dsname allocation)[2] |
| 0224 | (548) | Eligible device types do not contain enough units. (dsname allocation)[1] |
| 0228 | (552) | Specified volume or unit in use by system. (dsname allocation) |
| 022C | (556) | Volume mounted on ineligible permanently resident or reserved unit. (dsname allocation) |
| 0230 | (560) | Permanently resident or reserved volume on required unit. (dsname allocation) |
| 0234 | (564) | More than one device required for a request specifying a specific unit. (dsname allocation) |
| 0238 | (568) | Space unavailable in task input output table (TIOT). (dsname allocation, concatenation) |
| 023C | (572) | Required catalog not mounted, and user does not have volume mounting authorization. (dsname allocation) |
| 0240 | (576) | Requested device is a console. (dsname allocation) |
| 0244 | (580) | Telecommunication device not accessible. (dsname allocation) |
| 0248 | (584) | MSS virtual volume cannot be mounted. (dsname allocation) |
| 024C | (588) | Operating-system-managed resource was unavailable to the subsystem. (dsname allocation)[3] |
| 0250 | (592) | Subsystem resource not available. (dsname allocation)[3] |
| 0254 | (596) | The TIOT resource is currently unavailable and the user requested conditional ENQ on the resource. (all SVC 99 functions) |
| 0258 | (600) | There was not a sufficient number of non-restricted units to satisfy the request, or JES3 selected a JES3-managed restricted unit to satisfy the request. (dsname allocation) |
| 025C | (604) | 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) |
| 0260 | (608) | Unit does not meet specified status requirements. (dsname allocation) |
| 0264 | (612) | Invalid request due to current unit status. (dsname allocation) |
| 0268 | (616) | Tape device is broken. (dsname allocation) |
| 026C | (620) | Request requires more SMS-managed volumes than are eligible. |
| 0270 | (624) | Request requires more non-SMS-managed volumes than are eligible. |

[1] The conditions that cause these return codes are detected by MVS or JES3.

[2] For MSS requests, the MSSC reason code for this failing job step is contained in message IEF710I on the hardcopy log. An explanation of the MSSC reason code is contained in *Mass Storage System Extensions Messages*. For non-MSS requests, this code is accompanied by message IEF485I. It may result from a JES3 failure because of a busy or unavailable situation.

[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.

Figure 60. Class 2 Error Reason Codes (Unavailable System Resource)

| CLASS 3 CODES | | (INVALID PARAMETER LIST) |
|---|---|---|
| **Hex Code** | **(Decimal)** | **Meaning** |
| 0304-0338 | (772-824) | Assigned by DAIR. |
| 033C-0354 | (828 852) | Reserved. |
| 0358 | (856) | Overriding disposition of DELETE invalid for data set allocated as SHR. (unallocation)[1] |
| 035C | (860) | Invalid PARM specified in text unit. (all SVC 99 functions)[2] |
| 0360 | (864) | Invalid KEY specified in text unit. (all SVC 99 functions)[2] |
| 0364 | (868) | JOBLIB/STEPLIB/JOBCAT/STEPCAT specified as ddname, or associated with specified dsname. (dsname allocation, ddname allocation, unallocation, concatenation, deconcatenation)[1] |
| 0368 | (872) | Authorized function requested by unauthorized user. (all SVC 99 functions) |
| 036C | (876) | Invalid parameter list format. (all SVC 99 functions) |
| 0370 | (880) | Reserved. |
| 0374 | (884) | Invalid # specified in text unit. (all SVC 99 functions)[2] |
| 0378 | (888) | Duplicate KEY specified in text unit. (all SVC 99 functions)[2] |
| 037C | (892) | Invalid LEN specified in text unit. (all SVC 99 functions)[2] |
| 0380 | (896) | 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] |
| 0384 | (900) | Mutually inclusive KEY not specified. One key was used; two should have been used. (dsname allocation, unallocation)[2] |
| 0388 | (904) | Required key not specified. (ddname allocation, information retrieval, concatenation, deconcatenation, remove-in-use processing, unallocation) |
| 038C | (908) | Duplicate ddnames specified. (concatenation) |
| 0390 | (912) | GDG group name specified with relative generation number exceeds 35 characters. (dsname allocation) |
| 0394 | (916) | Status and relative generation number are incompatible. (dsname allocation) |
| 0398 | (920) | Volume sequence number exceeds the number of volumes. (dsname allocation) |
| 039C | (924) | Device type and volume are incompatible. (dsname allocation) |
| 03A0 | (928) | Subsystem detected an invalid parameter. (dsname allocation)[3] |
| 03A4 | (932) | Unable to protect data set/volume because of conflicting keyword specification. |
| 03A8 | (936) | Request block extension has invalid format. |
| 03AC | (940) | The CPPL address is not specified in request block extension |

[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.

Figure 61. Class 3 Error Reason Codes (Invalid Parameter List)

| CLASS 4 CODES | | (ENVIRONMENTAL ERROR) |
|---|---|---|
| **Hex Code** | **(Decimal)** | **Meaning** |
| 0404-040C | (1028-1036) | Reserved. |
| 0410 | (1040) | Specified ddname unavailable. (dsname allocation, ddname allocation) |
| 0414-041C | (1044-1052) | Reserved. |
| 0420 | (1056) | Specified ddname or dsname associated with an open data set. (ddname allocation, concatenation, deconcatenation, unallocation, dsname allocation)[1] |
| 0424 | (1060) | Deconcatenation would result in duplicate ddnames (deconcatenation).[1] |
| 0428-0430 | (1064-1072) | Reserved. |
| 0434 | (1076) | Ddname specified in ddname allocation request is associated with a convertible or non-permanently allocated resource. (ddname allocation) |
| 0438 | (1080) | Specified ddname not found. (information retrieval, ddname allocation, concatenation, deconcatenation, unallocation) |
| 043C | (1084) | The system could not deallocate enough of the resources being held in anticipation of reuse to meet the control limit. (dsname allocation) |
| 0440 | (1088) | Specified dsname not found. (information retrieval, unallocation) |
| 0444 | (1092) | Relative entry number specified in information retrieval request not found. (information retrieval) |
| 0448 | (1096) | Request for a new data set failed; the data set already exists. (dsname allocation) |
| 044C | (1100) | Request was made for a data set that has a disposition of delete; this request cannot be honored because the data set may be deleted at any time. (dsname allocation) |
| 0450 | (1104) | Request would cause the limit of 1635 concurrent allocations to be exceeded. (dsname allocation) |
| 0454 | (1108) | Ddname in DCB reference not found. (dsname allocation) |
| 0458 | (1112) | Dsname in DCB reference or volume reference is a GDG group name. (dsname allocation) |
| 045C | (1116) | Specified dsname to be deallocated is a member of a permanently-concatenated group. (unallocation)[1] |
| 0460 | (1120) | Specified dsname or member to be deallocated is not associated with specified ddname. (unallocation) |
| 0464 | (1124) | Specified dsname to be deallocated is a private catalog. (unallocation)[1] |

[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 MSSC reason code for this failing job step is contained in message IEF710I on the hardcopy log. An explanation of the MSSC reason code is contained in *Mass Storage System Extensions Messages.*

[3] This code corresponds to MSSC reason code '007', which is explained in *Mass Storage System Extensions Messages.*

[4] This code corresponds to MSSC reason code '207', which is explained in *Mass Storage System Extensions Messages.*

[5] 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.

[6] The information reason code field contains the MSSC reason code. An explanation of the MSSC reason code is contained in *Mass Storage System Extensions Messages.*

Figure 62 (Part 1 of 2). Class 4 Error Reason Codes (Environmental Error)

**CLASS 4 CODES, continued**

| Hex Code | (Decimal) | Meaning |
|---|---|---|
| 0468 | (1128) | Error while allocating or opening a private catalog. (dsname allocation) |
| 046C | (1132) | Remote work station not defined to job entry subsystem. (dsname allocation, unallocation) |
| 0470 | (1136) | User unauthorized for subsystem request. (dsname allocation) |
| 0474 | (1140) | Error while attempting to select optimum device. (dsname allocation). |
| 0478 | (1144) | Unable to process job entry subsystem request. (dsname allocation, unallocation) |
| 047C | (1148) | Unable to establish ESTAE environment. (all SVC 99 functions) |
| 0480 | (1152) | The number of units needed to satisfy the request exceeds the limit. (dsname allocation) |
| 0484 | (1156) | Request denied by operator. (dsname allocation) |
| 0488 | (1160) | GDG pattern DSCB not mounted. (dsname allocation) |
| 048C | (1164) | GDG pattern DSCB not found. (dsname allocation) |
| 0490 | (1168) | Error changing allocation assignments. (dsname allocation) |
| 0494 | (1172) | Error processing OS CVOL. (dsname allocation) |
| 0498 | (1176) | MSS virtual volume not accessible. (dsname allocation)[2] |
| 049C | (1180) | MSS virtual volume not defined. (dsname allocation)[3] |
| 04A0 | (1184) | Specified MSVGP name not defined. (dsname allocation)[4] |
| 04A4 | (1188) | Subsystem request in error. (dsname allocation)[5] |
| 04A8 | (1192) | Subsystem does not support allocation via key DALSSNM. (dsname allocation) |
| 04AC | (1196) | Subsystem is not operational. |
| 04B0 | (1200) | Subsystem does not exist. |
| 04B4 | (1204) | Protect request not processed; RACF not in system or not active. |
| 04B8 | (1208) | MSS not initialized for allocation. (dsname allocation) |
| 04BC | (1212) | MSS volume select error. (dsname allocation)[6] |
| 04C0 | (1216) | Protect request failed; user not defined to RACF. (dsname allocation) |
| 04C4 | (1220) | 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 972 backward references are allowed if the data set names are 44 characters in length.) |
| 04C8 | (1224) | A non-zero return code was set in register 15 from either common allocation or JFCB housekeeping; however, the SIOT reason code (SIOTRSNC) was not set. This problem might result from installation modification of the eligible device table (EDT). (dsname allocation) |
| 04CC | (1228) | Invalid output descriptor or invalid ddname reference. |
| 04D0 | (1232) | SMS (storage management subsystem) is not available. |

[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 MSSC reason code for this failing job step is contained in message IEF710I on the hardcopy log. An explanation of the MSSC reason code is contained in *Mass Storage System Extensions Messages*.

[3] This code corresponds to MSSC reason code '007', which is explained in *Mass Storage System Extensions Messages*.

[4] This code corresponds to MSSC reason code '207', which is explained in *Mass Storage System Extensions Messages*.

[5] 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.

[6] The information reason code field contains the MSSC reason code. An explanation of the MSSC reason code is contained in *Mass Storage System Extensions Messages*.

Figure 62 (Part 2 of 2). Class 4 Error Reason Codes (Environmental Error)

**CLASS 7 CODES** **(SYSTEM ROUTINE ERROR)**

**Note:** The failing system routine returns the code represented by "zz."

| Hex Code | Meaning |
|---|---|
| 17zz | LOCATE error. (Note: '08', '18', and '2C' 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) |
| 27zz | Reserved |
| 37zz | Reserved |
| 47zz | DADSM allocate error. (dsname allocation)[1] |
| 57zz | CATALOG error. (dsname allocation) |
| 67zz | OBTAIN error. (dsname allocation, information retrieval)[2] |
| 7700 | Subsystem error. (dsname allocation)[3] |
| 7704 | A subsystem interface system error occurred while processing key DALSSNM. |
| 8700 | Scheduler JCL Facility (SJF) error [4] |
| 8704 | Scheduler JCL Facility access function error |
| 8708 | Mutual exclusivity checker error [4] |
| 9700 | Severe SMS (storage management subsystem) IDAX error [5] |
| 9704 | Severe SMS CATALOG error [5] |
| 9708 | Severe SMS VOLREF error [5] |
| 970C | Severe SMS VTOC error [5] |
| 9710 | Severe SMS DISP error [5] |
| 9714 | Severe SMS COPY SWB error [5] |
| 9728 | System error while allocating a device [4] |

[1] DADSM return codes can be found in *Diagnosis Reference*. The information reason code field might contain a value that further describes the error. An explanation of this value is contained in the allocation message corresponding to the error code. See the section on reason codes in the allocation/unallocation component of *System Logic Library* for the message number associated with the error and *Message Library: System Messages* for the message itself.

[2] OBTAIN return codes can be found in *Diagnosis Reference*.

[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] A system error occurred.

[5] An error was detected by SMS. Request the message processing option of dynamic allocation to obtain messages relating to the error.

Figure 63. Class 7 Error Reason Codes (System Routine Error)

## SVC 99 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 accompanies the DYNALLOC macro. The text units are arranged according to the functions they request, in ascending order of their KEY values. See Figure 58 for a general description of the text unit and text unit keys.

You request a particular SVC 99 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 | SVC 99 Function |
|---|---|
| **'01'** | Dsname allocation |
| **'02'** | Unallocation |
| **'03'** | Concatenation |
| **'04'** | Deconcatenation |
| **'05'** | Remove-in-use processing based on task-ID |
| **'06'** | Ddname allocation |
| **'07'** | Information retrieval |

Figure 64 through Figure 72 present the SVC 99 text units in list form, introductory to the descriptions of the text units in each verb code group. The mnemonics given for the text units are those assigned by mapping macro IEFZB4D2.

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 their mnemonics, in Figure 64. The descriptions of the text units are in order of the key numbers, for easy reference.

Note that the values you specify in the text units are in binary (hexadecimal) and EBCDIC.

| Hex Text Unit Key | IEFZB4D2 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. |
| 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. |

Figure 64 (Part 1 of 2). Verb Code 01 (Dsname Allocation) - Text Unit Keys, Mnemonics, and Functions

| Hex Text Unit Key | IEFZB4D2 Mnemonic | Dsname Allocation Function |
|---|---|---|
| 0027 | DALQNAME | Names a TPROCESS macro, and a TCAM procedure. |
| 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 remote work station routing for the SYSOUT data set. |
| 0059 | DALSHOLD | Specifies hold queue routing for the SYSOUT data set. |
| 005E | DALMSVGP | Specifies a group of MSS virtual volumes. |
| 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 a user ID to which the SYSOUT data set is to be routed. |
| 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. |
| 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. |
| 8002 | DALOUTPT | Refers to a specific OUTPUT JCL statement. |
| 8004 | DASTCL | Specifies the storage class of a new SMS-managed data set. |
| 8005 | DAMGCL | Specifies the management class of a new SMS-managed data set. |
| 8006 | DADACL | Specifies the data class of a new SMS-managed data set. |
| 800B | DALRECO | Specifies the organization of a new VSAM data set. |
| 800C | DALKEYO | Specifies the key offset of a new VSAM data set. |
| 800D | DALREFD | Copies data set attributes from a DD statement. |
| 800E | DALSECM | Copies data set attributes from a RACF data set profile. |
| 800F | DALLIKE | Copies data set attributes from a model data set. |
| 8010 | DALAVGR | Specifies the value of the allocation unit. |

Figure 64 (Part 2 of 2). Verb Code 01 (Dsname Allocation) - Text Unit Keys, Mnemonics, and Functions

## 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 described on the following pages and listed in Figure 64. The text units that represent DCB attributes are described under "DCB Attribute Text Units" and listed in Figure 65. The meaning of the parameters is the same as when specified on a DD statement as described in *JCL User's Guide* and *JCL Reference*.

For dsname allocation text units that do not have a JCL equivalent, see Figure 66 and the topic "Non-JCL Dsname Allocation Functions."

### 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 parameter field, 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. Dynamic allocation does not support backward references. See Figure 57. 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.

**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. 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:

'01' if OLD is desired
'02' if MOD is desired
'04' if NEW is desired
'08' if SHR is desired

**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:

'01' if UNCATLG is desired
'02' if CATLG is desired
'04' if DELETE is desired
'08' if KEEP is desired

**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'

DALBLKLN specifies the average data block length to be used by the system in computing the amount of space to allocate. The primary quantity space key (DALPRIME) or the secondary quantity space key (DALSECND) must also be specified when you code this key. The text unit keys that request space in terms of tracks (DALTRK) or cylinders (DALCYL, DALROUND) are mutually exclusive with DALBLKLN. When you code this key, # must be one, LEN must be three, and PARM contains the average data block length. The maximum PARM value is '00FFFF' (65,535). DALBLKLN is mutually exclusive with DALAVGR.

**Example:** to specify an average data block 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 unused space is to be deleted when the data set is closed. When you code this key, # must be zero; LEN and PARM are not 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:

'02' if different areas of contiguous space are to be allocated (ALX)
'04' if maximum contiguous space is required (MXIG)
'08' if 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.

**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 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.) 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
```

### Unit Description Specification - Key = '0015'

DALUNIT specifies a unit as a group (esoteric) name, a device type (generic), or a specific unit address (in EBCDIC). When you code DALUNIT, # must be one, LEN is the actual length of the unit description, and PARM contains the unit description.

**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 3330, code

```
KEY    #      LEN    PARM
0015   0001   0004   F3 F3 F3 F0
```

**Example:** to specify the unit address 230, code

```
KEY    #      LEN    PARM
0015   0001   0003   F2 F3 F0
```

### Unit 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 unit count. The maximum PARM value is '3B' (59).

**Example:** to specify a unit 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 obtain the class from the OUTPUT DD statement, if specified, code zero in the # field; LEN and PARM are not specified. If no OUTPUT DD statement is found, the default message class is used. 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

**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 actual length of the name, and PARM contains the program name.

**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 actual length of the form number, 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.

**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:

'01' if the volume has no label (NL)
'02' if the volume has an IBM standard label (SL)
'04' if the volume has a non-standard label (NSL)
'0A' if the volume has both an IBM standard label and a user label (SUL)
'10' if label processing is to be bypassed (BLP)
'21' if the system is to check for and bypass a leading tape mark on DOS unlabeled tape (LTM)
'40' if the volume has an American National Standard label (AL)
'48' if 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:

'10' if the data set should not be read, changed, extended, or deleted without the password.

'30' if 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 *JCL User's Guide* and *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:

'40' if output only is to be requested.
'80' if 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 retention period (DALRETPD) and SYSOUT (DALSYSOU) keys. 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 must be two, and PARM contains the retention period. The maximum PARM value is '270F' (9999).

**Example:** to specify a retention period of 10 days, code

```
KEY   #     LEN   PARM
0023  0001  0002  000A
```

### 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. 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:

'04' if verification is requested (VERIFY).
'08' if 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 and an optional qualifier designating the particular TCAM job or started task associated with the process name. 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") 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
```

**Example:** to specify the process name TPR.TCAM2, code

```
KEY    #      LEN     PARM
0027   0001   0009    E3 D7 D9 4B E3 C3 C1 D4 F2
```

### 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") 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, RKP, 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). When DALDCBDS is specified, # must be one, LEN is the length of the dsname, and PARM contains the data set same. (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'

DALSUSER requests that the SYSOUT data set being allocated be routed to a remote work station when it is deallocated. When coded in conjunction with the user ID key (DALUSRID), this key represents the node to which the user ID is assigned. The SYSOUT key (DALSYSOU) is required with this key. When you code DALSUSER, # must be one, LEN is the length of the work station name (maximum of 8), and PARM contains the work station name.

**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. When you code this key, # must be zero; LEN and PARM are not specified.

**Example:** to specify hold, code

```
KEY    #      LEN    PARM
0059   0000   -      -
```

### MSVGP Specification - Key = '005E'

DALMSVGP specifies a group of MSS virtual volumes. It is mutually exclusive with the SYSOUT (DALSYSOU), QNAME (DALQNAME), and volume serial (DALVLSER) text unit keys. When you code DALMSVGP, # must be one, LEN is the actual length of the MSVGP name, and PARM contains the group name.

**Example:** to specify a MSS volume group of SYSGROUP, code

```
KEY    #      LEN    PARM
005E   0001   0008   E2 E8 E2 C7 D9 D6 E4 D7
```

### Subsystem Name Request Specification - Key = '005F'

DALSSNM specifies a subsystem data set. 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 or national. See *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 LEN and PARM combinations that are present (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
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 *JCL User's Guide* and *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'

DALUSRID requests that the SYSOUT data set being allocated be routed to the specified user ID at a remote location. The SYSOUT (DALSYSOU) and SYSOUT remote work station (DALSUSER) keys are required with this key. The SYSOUT program name key (DALSPGNM) is mutually exclusive with DALUSRID.

When you code this key, # must be one, LEN is the length of the user ID (maximum of 8), and PARM contains the user ID itself. 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:

'02' for burster-trimmer-stacker
'04' for 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, LEN contains the length of the immediately-following character arrangement table, 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, 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 immediately-following form name, 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 immediately-following module name 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:

'00' for the first character arrangement table specified on the DALCHARS text unit
'01' for the second character arrangement table specified
'02' for the third character arrangement table specified
'03' for 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 *JCL User's Guide* and *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. 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
```

### OUTPUT Statement Reference - Key = '8002'

DALOUTPT explicitly associates a SYSOUT data set with the OUTPUT JCL statement specified in the PARM field. The SYSOUT (DALSYSOU) text unit key is required with this key.

When you code DALOUTPT, # is a number ranging from one to 128 ('0080'), LEN is the length of the PARM field, (maximum of '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
8002   0002   000A  E2 E3 C5 D7 F1 4B D6 E4 E3 F1
              0004  D6 E4 E3 E7
```

### Storage Class Specification - Key = '8004'

Code DALSTCL to specify the storage class of an SMS-managed data set.

**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'

Code DALMGCL to specify the management class of an SMS-managed data set.

**Example:** To specify the management class of "SAM" SMS-managed data set, code

```
KEY    #      LEN    PARM
8005   0001   0003   E2 C1 D4
```

### Data Class Specification - Key = '8006'

Code DALDACL to specify the data class of the data set.

**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'

Code DALRECO to specify the record organization of a VSAM data set.

- For a VSAM key-sequenced data set, code X'80'.
- For a VSAM entry-sequenced data set, code X'40'.
- For a VSAM relative record data set, code X'20'.
- For a VSAM linear space data set, code X'10'.

**Example:** To specify a key-sequenced record organization, code

```
KEY    #      LEN    PARM
800B   0001   0001   80
```

### Key Offset Specification - Key = '800C'

Code DALKEYO to specify 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.

**Example:** To specify a key offset of 18 decimal (12 hexadecimal) bytes, code

```
KEY    #      LEN    PARM
800C   0001   0001   12
```

### Copy DD Specification - Key = '800D'

Code DALREFD to specify the name of the JCL DD statement from which the attributes are to be copied.

**Example:** To copy the data set attributes from the JCL DD statement named "SAM", code

```
KEY    #      LEN    PARM
800D   0001   0003   E2 C1 D4
```

The specified name must be left justified in 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. Place the length of the name in LEN, and the value, X'0001', into the # field.

### Copy Profile Specification - Key = '800E'

Code DALSECM to specify the name of the RACF profile from which the RACF profile is to be copied.

**Example:** To copy the generic RACF profile, "RPROF", code

```
KEY    #      LEN1   PARM1              LEN2   PARM2
800E   0002   0005   D9 D7 D9 D6 C6     0001   80
```

If the copied profile is generic, such as in the JCL statement, SECMODEL=(*RPROF*,GENERIC), place the dsname value in PARM1 and its length in LEN1. Place X'80' in PARM2 and X'0001' in LEN2. Also, place X'0002' in the # field to indicate that the profile is generic. Define PARM2 as one byte, and PARM1 for as many bytes as required to hold the name.

If the copied profile was not defined generically, do not define LEN2 or PARM2. Place the dsname in PARM1, and its length into LEN1. Place X'0001' in the # field.

### Copy Model Specification - Key = '800F'

Code DALLIKE to specify the name of the model data set from which the attributes are to be copied.

**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'

Code DALAVGR to specify the allocation unit to be used when the data set is allocated. Code PARM as X'80', X'40', or X'20'. Code the LEN and the # fields as X'0001'.

- 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
```

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 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 | Specifies the buffer count per GET macro. |
| 0038 | DALBUFSZ | Specifies the line group buffer size. |
| 0039 | DALCODE | Specifies the data's paper tape code. |
| 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. |
| 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 | Specifies the user's telecommunications input work areas size. |
| | | 004D DALSTACK Specifies the card punch's stacker bin. |
| 004E | DALTHRSH | Specifies the use percentage of nonreusable direct access message queue records per flush closedown. |
| 004F | DALTRTCH | Specifies the 7-track tape recording technique. |
| 0051 | DALIPLTX | Specifies a TCAM network control program name. |
| 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 | Specifies input to the 3886 Character Reader. |

Figure 65. Verb Code 01 (DCB Attributes) – Text Unit Keys, Mnemonics, and Functions

## DCB Attribute Text Units

Use verb code 01 and the text unit keys listed in Figure 65 and described on the following pages to specify the DCB attributes of the data set being dynamically allocated. These attributes are described in *JCL User's Guide* and *JCL Reference* under the DCB parameter, and in *Data Administration: Macro Instruction Reference*.

### 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:

'01' for fullword not a doubleword boundary (F)
'02' for 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:

'08' for dynamic buffering (D)
'10' for exchange buffering (E)
'20' for record buffering (R)
'40' for simple buffering (S)
'60' for 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, and PARM contains the block size. The maximum PARM value is '7FF8' (32,760).

**Example:** to specify a block size of 80, code

```
KEY    #      LEN    PARM
0030   0001   0002   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:

'80' the block prefix is four bytes long and contains the block length (L)
'nn' the length of the block prefix (maximum of '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 instruction. 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 blocksize 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:

'02' for Teletype 5-track (T)
'04' for USASCII 8-track (A)
'08' for National Cash Register 8-track (C)
'10' for Burroughs 7-track (B)
'20' for Friden 8-track (F)
'40' for IBM BCD 8-track (I)
'80' for 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:

'01' for send priority (S)
'02' for equal priority (E)
'04' for 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:

'03' for 200 bpi 7-track (0)
'43' for 556 bpi 7-track (1)
'83' for 800 bpi 7-track, 800 bpi 9 - track (2)
'C3' for 1600 bpi 9-track (3)
'D3' for 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:

'0004' for TCAM 3705
'0008' for VSAM
'0020' for TCAM message queue (TQ)
'0040' for TCAM line group (TX)
'0080' for graphics (GS)
'0200' for partitioned organization (PO)
'0300' for partitioned organization unmovable (POU)
'0400' for government of message transfer to or from a telecommunications message processing queue (MQ)
'0800' for direct access message queue (CQ)
'1000' for communication line group (CX)
'2000' for direct access (DA)
'2100' for direct access unmovable (DAU)
'4000' for physical sequential (PS)
'4100' for 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:

'10' for online BSAM testing (T)
'20' to cause abnormal end of task (ABE)
'40' to skip the block causing the error (SKP)
'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:

'8000' for variable length spanned records processed under QSAM and BSAM, the logical records exceed 32,756 bytes (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:

'40' for EBCDIC mode (E)
'50' for EBCDIC, read column eliminate mode (ER)
'60' for EBCDIC, optical mark read mode (EO)
'80' for card image mode (C)
'90' for card image, read column eliminate mode (CR)
'A0' for 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 '63' (99).

**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 by the control program. When you code this key, # and LEN must be one, and PARM contains one of the following values:

'01' for relative block addressing (R), or to select character arrangement tables for the 3800 printer (J)

'02' for user totaling facility (T)

'04' for reduced tape error recovery or direct DASD search (Z)

'08' for direct addressing (A), or
for translation of ASCII to or from EBCDIC (Q)

'10' for feedback (F), or
for hopper-empty exit (H), or
for online correction for optical readers (O)

'20' for chained scheduling or TCAM segment identification (C), or
for extended search (E)

'40' for disregarding end-of-file recognition for tape (B), or
for allowance of data checks caused by an invalid character, or
for handling a TCAM work unit as a message (U)

'80' for write validity check, or to place TCAM message source in an eight-byte field in the workarea (W)

**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 *Data Administration: Macro Instruction Reference*.

**Example:** to specify OPTCD value U, code

```
KEY    #      LEN    PARM
0045   0001   0001   40
```

**Example:** to specify OPTCD values U and C, code

```
KEY    #      LEN    PARM
0045   0001   0001   60
```

### 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:

'02' for a PCI and no new buffer allocated (R)
'08' for no PCIs (N)
'20' for a PCI and new buffer allocated (A)
'80' for 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 this key is specified, # and LEN contain one, and PARM contains:

'01' for a PCI and no new buffer allocated (R)
'04' for no PCIs (N)
'10' for a PCI and a new buffer allocated (A)
'40' for 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:

'01' for no spacing (0)
'09' for one-line spacing (1)
'11' for two-line spacing (2)
'19' for 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:

'02' for machine code printer control characters in record (M), or
for complete QTAM record (R)

'04' for ASA printer control characters in record (A), or
for complete QTAM message (G)

'08' for standard fixed records, spanned variable records, or
segment of QTAM message (S)

'10' for blocked records (B)

'20' for variable ASCII records (D), or
for track overflow (T)

'40' for variable records (V)

'80' for fixed records (F)

'C0' for 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:

'01' for bin 1 (1)
'02' for 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 nonreusable 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:

'13' for data conversion (C)
'23' for even parity (E)
'2B' for even parity and BCD/EBCDIC translation (ET)
'3B' for BCD/EBCDIC translation (T)

**Example:** to specify even parity, code

```
KEY    #      LEN    PARM
004F   0001   0001   23
```

### IPLTXTID Specification - Key = '0051'

DALIPLTX specifies the name of a TCAM network control program. It is mutually exclusive with the DSNAME (DALDSNAM), MEMBER NAME (DALMEMBR), and QNAME (DALQNAME) keys. When you code DALIPLTX, # must be one, LEN is the length of the name (maximum of 8), and PARM contains the name.

**Example:** to specify an IPLTXTID value of PGM, code

```
KEY    #      LEN    PARM
0051   0001   0003   D7 C7 D4
```

### 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.

**Note:** 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:

'10' for W
'12' for WT
'14' for WX
'16' for WXT
'20' for P
'30' for PW
'34' for PWX
'36' for PWXT
'40' for R
'50' for RW
'52' for RWT
'54' for RWX
'56' for RWXT
'60' for RP
'68' for RPD
'70' for RPW
'74' for RPWX
'76' for RPWXT
'78' for RPWD
'80' for 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

**Notes:**

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
```

## Non-JCL Dynamic Allocation Functions

The keys listed in Figure 66 and described on the following pages do not have JCL equivalents; they have meaning only to the SVC 99 routines in performing dynamic allocation by dsname (verb code 01). You can specify the information retrieval keys C004-C010 when performing a dynamic allocation by both ddname and dsname.

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 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. |

Figure 66. Verb Code 01 (Non-JCL Dsname Functions) – Text Unit Keys, Mnemonics, and Functions

### 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, 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. 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.

**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 SVC 99. When you code DALRTDDN, # must be one, LEN must be eight, and PARM is an eight-byte field. The SVC 99 routines place the allocated ddname in PARM and update LEN to the length of the 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 SVC 99. When you code DALRTDSN, # must be one, LEN must be forty-four, and PARM is a forty-four byte field. The SVC 99 routines place the allocated dsname in PARM and update LEN to the length of the 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 SVC 99. When you code DALRTORG, # must be one, LEN must be two, and PARM is a two-byte field. The SVC 99 routines put one of the following values into PARM:

'0000' if the dynamic allocation routines cannot determine the DSORG
'0004' if TR
'0008' if VSAM
'0020' if TQ
'0040' if TX
'0080' if GS
'0200' if PO
'0300' if POU
'0400' if MQ
'0800' if CQ
'1000' if CX
'2000' if DA
'2100' if DAU
'4000' if PS
'4100' if PSU
'8000' if IS
'8100' if ISU

**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 name (maximum of 4), and PARM contains the subsystem name.

**Note:** To specify DALSSREQ, your program must be APF-authorized, in supervisor state, or running in a system protection key.

**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, the SVC 99 routines put the number in PARM. If the volume serial is not available at the completion of allocation, the SVC 99 routines set 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.

**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 in a system protection key.

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
```

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 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. |
| 0007 | DUNUNALC | Specifies deallocation even if the resource has the permanently allocated attributed. |
| 0008 | DUNREMOV | Specifies removal of the "in-use" attribute, even if the resource has the permanently allocated attribute. |
| 000A | DUNOVSNH | Specifies "nohold" status for a deallocated SYSOUT data set. |
| 0018 | DUNOVCLS | Specifies an overriding SYSOUT class. |
| 0058 | DUNOVSUS | Specifies an overriding remote workstation. |
| 0059 | DUNOVSHQ | Puts the SYSOUT data set on the hold queue and overrides previous "nohold" specifications. |

Figure 67. Verb Code 02 (Dynamic Unallocation) – Text Unit Keys, Mnemonics, and Functions

## Dynamic Unallocation Text Units

Use verb code 02 and the text unit keys listed in Figure 67 and described on the following pages to request dynamic unallocation processing by the SVC 99 routines.

### 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, 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. When you code this key, # must be one, LEN contains the length of the dsname, and PARM contains the dsname.

**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 dsname unallocation key (DUNDSNAM) is required with this key. When you code DUNMEMBR, # must be one, LEN is the 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
```

### 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:

'01' for an overriding disposition of UNCATLG
'02' for an overriding disposition of CATLG
'04' for an overriding disposition of DELETE
'08' for an overriding disposition of KEEP

**Example:** to specify an overriding disposition of CATLG, code

```
KEY    #      LEN    PARM
0005   0001   0001   02
```

**Note:** The SVC 99 routines ignore 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 the SVC 99 routines must ignore a DUNOVDSP request, they still perform the deallocation processing, but use the disposition from the original allocation request.

### Unalloc Option Specification - Key = '0007'

DUNUNALC specifies that the resource is to be deallocated even if it has the permanently allocated attribute. The remove 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 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. 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.

**Example:** to specify an overriding class of C, code

```
KEY    #      LEN    PARM
0018   0001   0001   C3
```

### Overriding SYSOUT Remote Workstation Specification - Key = '0058'

DUNOVSUS specifies that the SYSOUT data set being deallocated is to be routed to a remote user. This specification overrides the remote workstation specification assigned when the data set was allocated. DUNOVSUS is ignored if the data set is not a SYSOUT data set.

When you code DUNOVSUS, # must be one, LEN is the length of the remote workstation name (maximum of 8), and PARM contains the remote workstation (user) name.

**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 is 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. 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   -      -
```

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 Function |
|---|---|---|
| 0001 | DCCDDNAM | Specifies the ddnames to be concatenated. |
| 0004 | DCCPERMC | Specifies the permanently concatenated attribute. |

Figure 68. Verb Code 03 (Dynamic Concatenation) – Text Unit Keys, Mnemonics, and Functions

## Dynamic Concatenation Text Units

Use verb code 03 and the text units listed in Figure 68 and described in the following paragraphs to request dynamic concatenation processing by the SVC 99 routines.

### 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 (a minimum of two), LEN is the length of the immediately following ddname, 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. 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   -      -
```

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 Function |
|---|---|---|
| 0001 | DDCDDNAM | Specifies the ddname of the group to be deconcatenated. |

Figure 69. Verb Code 04 (Dynamic Deconcatenation) – Text Unit Key, Mnemonic, and Function

## Dynamic Deconcatenation Text Unit

Use verb code 04 and the following text unit to request dynamic deconcatenation processing by the SVC 99 routines.

### 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 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
```

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 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 but those of the current task and its higher-level tasks. |

Figure 70. Verb Code 05 (Remove-In-Use Processing Based on Task-ID) – Text Unit Keys, Mnemonics, and Functions

## Text Units for Removing the In-Use Attribute Based on Task-ID

Use verb code 05 and the text units in Figure 70 and described as follows to request that the SVC 99 routines turn off the in-use bits for resources based on task-ID.

### 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 except those associated with the current task, its direct ancestors, and the initiator. 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   -      -
```

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 Function |
|---|---|---|
| 0001 | DDNDDNAM | Specifies the ddname to be allocated. |
| 0002 | DDNRTDUM | Requests a dummy data set indication. |

Figure 71. Verb Code 06 (Ddname Allocation) – Text Unit Keys, Mnemonics, and Functions

## Ddname Allocation Text Units

Use verb code 06 and the text units listed in Figure 71 and described as follows to request ddname allocation processing. In ddname allocation, you are specifying that the SVC 99 routines are 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.

### 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, 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. The SVC 99 routines set PARM as follows:

'80' if the ddname is associated with a dummy data set
'00' otherwise

**Example:** to specify that the DUMMY indication be returned, code

```
KEY    #      LEN    PARM
0002   0001   0001   -
```

| Hex Text Unit Key | IEFZB4D2 Mnemonic | SVC 99 Function |
|---|---|---|
| 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 return of the associated ddname. |
| 0005 | DINRTDSN | Requests the return of the data set name. |
| 0006 | DINRTMEM | Requests the return of the PDS member name. |
| 0007 | DINRTSTA | Requests the return of the data set's status. |
| 0008 | DINRTNDP | Requests the return of the data set's normal disposition. |
| 0009 | DINRTCDP | Requests the return of the data set's conditional disposition. |
| 000A | DINRTORG | Requests the return of 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 return of special attribute indications. |
| 000D | DINRTLST | Requests the return of a last relative entry indication. |
| 000E | DINRTTYP | Requests the return of the data set's type (terminal or dummy). |
| 000F | DINRELNO | Specifies the desired allocation information retrieval by relative request number. |
| 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. |

Figure 72. Verb Code 07 (Dynamic Information Retrieval) – Text Unit Keys, Mnemonics, and Functions

## Dynamic Information Retrieval Text Units

Use verb code 07 and the text units listed in Figure 72 and described as follows to request that the SVC 99 routines return certain information about the allocated resources.

### 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) and relative entry (DINRELNO) keys. When you code DINDDNAM, # must be one, LEN is the length of the ddname, 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'

DINDSNAM specifies the dsname of the allocated resource you are requesting information about. It is mutually exclusive with the ddname (DINDDNAM) and relative entry (DINRELNO) keys. When you code DINDSNAM, # must be one, LEN is the length of the dsname, and PARM contains the dsname.

**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. When you code this key, # must be one, LEN must be eight, and PARM is an eight-byte field. 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 forty-four, and PARM is a forty-four byte field. 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 eight, and PARM is an eight-byte field. Upon return to your program, PARM will contain the member name and LEN will be set to its length (or to zero, if none).

**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:

'01' for OLD
'02' for MOD
'04' for NEW
'08' for 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:

'01' for UNCATLG
'02' for CATLG
'04' for DELETE
'08' for KEEP
'10' for 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. The values for #, LEN and PARM are the same as for the return normal disposition key (DINRTNDP).

**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:

'0000' if undetermined
'0004' if TR
'0008' for VSAM
'0020' if TQ
'0040' if TX
'0080' for GS
'0200' for PO
'0300' for POU
'0400' for MQ
'0800' for CQ
'1000' for CX
'2000' for DA
'2100' for DAU
'4000' for PS
'4100' for PSU
'8000' for IS
'8100' for 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 resources that must be deallocated before a new allocation can be made. 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 to be deallocated.

**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, the SVC 99 routines return 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 reserved

**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 number 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:

'80' if last relative entry
'00' otherwise

**Example:** to request the return of the last entry, 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:

'80' if a DUMMY data set
'40' if a terminal allocation
'20' if a SYSIN data set
'10' if a SYSOUT data set
'00' otherwise

```
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) and dsname (DINDSNAM) 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 SVC 99 request, code

```
KEY    #      LEN    PARM
000F   0001   0002   000A
```

### Return Storage Class Specification - Key = 'C004'

Code DINRSTCL to request the storage class of the specified SMS-managed data set.

**Example:** What is the storage class of the specified SMS-managed data set?

```
KEY    #      LEN    PARM
C004   0001   0008   - - - - - - - -
```

The system returns the storage class identifier, left justified, into PARM, which must be an eight byte field. It also stores a two-byte number in LEN that indicates the length of the identifier.

### Return Management Class Specification - Key = 'C005'

Code DINRMGCL to request the management class of the specified SMS-managed data set.

**Example:** What is the management class of the specified SMS-managed data set?

```
KEY    #      LEN    PARM
C005   0001   0008   - - - - - - - -
```

The system returns the management class identifier, left justified, into PARM, which must be an eight-byte field. It also stores a two-byte number in LEN that indicates the length of the identifier.

### Return Data Class Specification - Key = 'C006'

Code DINRDACL to request the data class of the specified SMS-managed data set.

**Example:** What is the data class of the specified data set?

```
KEY    #      LEN    PARM
C006   0001   0008   40 40 40 40 40 40 40 40
```

The system returns the data class identifier, left justified, into PARM, which must be an eight-byte field. It also stores a two-byte number in LEN that indicates the length of the identifier.

### Return Record Organization Specification - Key = 'C00B'

Code DINRRECO to request the organization of the records in the specified VSAM data set.

**Example:** How are the records organized in the specified VSAM data set?

```
KEY    #      LEN    PARM
C00B   0001   0001   00
```

The system returns the record organization into PARM, which must be a one-byte field; the system also returns X'0001' into both the LEN field and the # field. The value returned in PARM is one of the following:

- For a VSAM key-sequenced data set (KS), X'80'
- For a VSAM entry-sequenced data set (ES), X'40'
- For a VSAM relative record data set (RR), X'20'
- For a VSAM linear space data set (LS), X'10'

### Return Key Offset Specification - Key = 'C00C'

Code DINRKEYO to request 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.

**Example:** What is the key offset in a record of the specified VSAM data set?

```
KEY    #      LEN    PARM
C00C   0001   0002   00 00
```

The system returns a two-byte binary number representing the offset into PARM, which must be a two-byte field. It stores X'0002' into LEN. The value of the offset is less than or equal to 65535 bytes.

### Return Copy DD Specification - Key = 'C00D'

Code DINRREFD to request the name of the JCL DD statement from which the attributes of the specified data set were copied.

**Example:** What is the name of the JCL DD statement from which the attributes of the specified data set were copied?

```
KEY    #      LEN    PARM
C00D   0001   001A   - - - ...
```

The system returns the name, left justified, into PARM, which you must define as a 26-byte field. It also stores the length of the name in LEN. 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.

### Return Copy Profile Specification - Key = 'C00E'

Code DINRSECM to request the name of the RACF profile from which the RACF profile of the specified data set was copied.

**Example:** What is the dsname of the RACF profile that was used to supply the profile of the specified data set?

```
KEY    #      LEN1   PARM1        LEN2   PARM2
C00E   0001   002C   - - - ...    0000   - - - - - - - -
```

If the copied profile was defined generically, such as in the JCL statement, SECMODEL=(*dsname*,GENERIC), the system returns the dsname value in PARM1 and the length of the dsname in LEN1. It stores X'80' in PARM2 and X'0001' in LEN2. It also stores X'0002' in the # field, indicating that the profile is generic. You must define PARM2 as one byte, and PARM1 as 44 bytes.

If the copied profile was not defined generically, the system does not store anything in LEN2 or PARM2. However, the system returns the dsname into PARM1, and the length of the dsname into LEN. The # field is set to X'0001', indicating that the profile is not generic.

### Return Copy Model Specification - Key = 'C00F'

Code DINRLIKE to request the name of the model data set from which the attributes of the specified data set were copied.

**Example:** What is the dsname of the model data set from which the attributes of the specified data set were copied?

```
KEY    #      LEN    PARM
C00F   0001   002C   - - - ...
```

The system returns the dsname into PARM, which you must define as a 44-byte field, and the length of the dsname into LEN. It stores X'0001' in the # field.

### Return Average Record Specification - Key = 'C010'

Code DINRAVGR to request the allocation unit that was used when the specified data set was allocated.

**Example:** What unit of allocation was used to allocate the specified data set?

```
KEY    #      LEN    PARM
C010   0001   0001   - - - ...
```

The system returns a code into PARM, which you must define as a one-byte field. The returned code is one of the following:

- U (X'80') represents single-record units.
- K (X'40') represents thousand-record units.
- M (X'20') represents million-record units

The system also returns X'0001' in the # field.

## Example of a Dynamic Allocation Request

The assembler language example in Figure 73 is a dynamic allocation request allocating SYS1.LINKLIB with a status of SHARE. It also requests that the SVC 99 routines return the ddname associated with SYS1.LINKLIB.

Figure 74 shows the parameter list that is built from the SVC 99 invocation in Figure 73.

```
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 SVC 99 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 73. Example of a Dynamic Allocation Request

Note the concepts that the example illustrates:

- You need to request storage via the GETMAIN macro for the request block and the DALRTDDN text unit, because the SVC 99 routines modify them. The DALDSNAM and DSLSTATS text units can be in virtual storage.

  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.

Figure 74 shows the parameter list that results from the code in Figure 73. It is the SVC 99 request block structure needed to allocate data set SYS1.LINKLIB with a disposition of SHARE, and to return the ddname assigned by SVC 99.

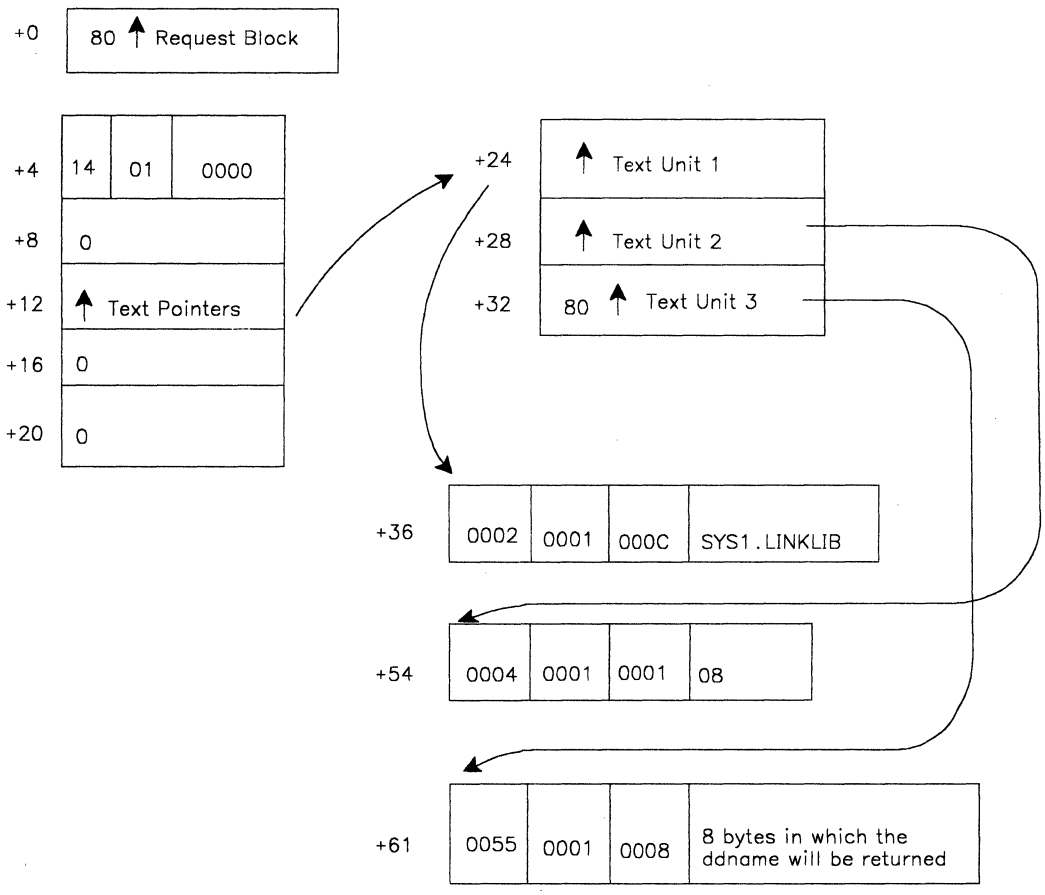



Figure 74. Parameter List Resulting From Dynamic Allocation Example

# Index

## A

## B



## C

## D

## E

## F



## G

## H



## I

## J



## K

# L

## M

## Q

# R

## S

# T

## U



## V

## W



## X



## Numerics

MVS/Extended Architecture System Programming Library: System Macros and Facilities Volume 1

GC28-1150-4 S370-36

Printed in U.S.A.

IBM®

MVS/Extended Architecture
System Programming
Library: System Macros
and Facilities
Volume 1

READER'S
COMMENT
FORM

GC28-1150-4

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

**Note:** ***Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.***

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

What is your occupation? _____

How do you use this publication? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

MVS/Extended Architecture System Programming Library: System Macros and Facilities Volume 1

GC28-1150-4 S370-36

Reader's Comment Form

Cut or Fold Along Line

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO Box 950
Poughkeepsie, New York 12602

Fold and tape Please Do Not Staple Fold and tape

Printed in U.S.A.

IBM®

MVS/Extended Architecture
System Programming
Library: System Macros
and Facilities
Volume 1

READER'S
COMMENT
FORM

GC28-1150-4

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

**Note:** ***Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.***

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

What is your occupation? _____

How do you use this publication? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

MVS/Extended Architecture System Programming Library: System Macros and Facilities Volume 1

GC28-1150-4 S370-36

Reader's Comment Form

Cut or Fold Along Line

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO Box 950
Poughkeepsie, New York 12602

Fold and tape Please Do Not Staple Fold and tape

Printed in U.S.A.

IBM®

GC28-1150-04