GC28-1154-4
File No. S370-36

Program Product

MVS/Extended Architecture Supervisor Services and Macro Instructions

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, GC28-1154-3. 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 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.

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, N.Y. 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

This book, intended mainly for the programmer coding in assembler language, describes how to use the services of the supervisor, the macro instructions used to request these services, and the linkage conventions used by the control program to provide these services.

The system programmer interested in additional information on the supervisor should see *MVS/Extended Architecture System Programming Library: System Macros and Facilities Volume 1*, GC28-1150 and *Volume 2*, GC28-1151.

About This Book

This book is divided into two parts. Part I, "Supervisor Services," provides explanations and aids for using the facilities available through the supervisor. Part II, "Macro Instructions," provides coding information.

Part I is divided into nine topics. Specific topics include:

- Linkage Conventions
- Subtask Creation and Control
- Program Management
- Resource Control
- Program Interruption, Termination, and Dumping Services
- Virtual Storage Management
- Real Storage Management
- Data in Virtual Facility
- Timing and Communication

Part II contains the descriptions and definitions of the supervisor macro instructions available in the assembler language. It provides applications programmers coding the assembler language with the information necessary to code the macro instructions. The standard, list, and execute forms of the macro instructions are grouped, where applicable, for ease of reference.

Trademarks

The following are trademarks of International Business Machines Corporation.

- MVS/DFP™
- MVS/XA™

Related Publications

Use of this book requires a basic knowledge of the operating system and of assembler language. Books that contain basic information are:

Assembler H Version 2 Application Programming: Language Reference, GC26-4037

IBM/370 Vector Operations, SA22-7125

MVS/Extended Architecture Checkpoint/Restart User's Guide, GC26-4012

MVS/Extended Architecture Data Administration Macro Instruction Reference, GC26-4014

MVS/Extended Architecture Data Administration Guide, GC26-4013

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 Linkage Editor and Loader User's Guide, GC26-4011

MVS/Extended Architecture Message Library: Routing and Descriptor Codes, GC28-1194

MVS/Extended Architecture Operations: JES3 Commands, SC23-0063

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

OS/VS2 MVS Planning: Global Resource Serialization, GC28-1062

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 1, GC28-1150 and *Volume 2*, GC28-1151

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

370-Extended Architecture: Principles of Operation, GA22-7085

MVS/Extended Architecture: Integrated Catalog Administration: Access Method Services Reference, GC26-4135

Note: All references to Assembler H in this publication indicate the program product Assembler H Version 2 (5668-962).

Contents

Figures

Summary of Amendments

Summary of Amendments
for GC28-1154-4
for MVS/System Product Version 2 Release 2.3

This major revision supports MVS/System Product Version 2 Release 2.3 and modifications required in conjunction with MVS/DFP Version 3.1. These changes include LOCVIEW, which is a new parameter of the DIV macro instruction.

The RACF macro instructions FRACHECK, RACHECK, RACROUTE, and RACSTAT have been moved to *MVS/XA SPL: System Macros and Facilities.*

This revision also contains minor technical and editorial changes.

Summary of Amendments
for GC28-1154-3
for MVS/System Product Version 2 Release 2

This revision contains changes to the SNAP, DOM, and ATTACH macro instructions, and a description of the new DIV macro instruction.

Summary of Amendments
for GC28-1154-2
for the following:
- MVS/System Product Version 2
Release 1.3 Vector Facility Enhancement
- RACF Version 1 Release 7
- PTF UZ90404

In support of RACF Version 1 Release 7, this revision contains changes to the FRACHECK, RACHECK, RACROUTE, and RACSTAT macro instructions.

In support of MVS/System Product Version 2, Release 1.3 Vector Facility Enhancement, this revision contains changes to the ESPIE, SNAP, and SPIE instructions for the Vector Facility.

In support of PTF UZ90404, this revision contains changes to the ATTACH, LINK, LOAD, and XCTL macro instructions.

The revision also contains minor technical and editorial changes.

Part I: Supervisor Services

MVS/XA Supervisor Services and Macro Instructions describes the operating system services that an unauthorized program can use. An unauthorized program is one that does not run in supervisor state, or have PSW key 0-7, or reside in an APF-authorized library. To use a service, the program issues a macro instruction.

The book consists of Part I and Part II. The first topic in Part I describes the linkage conventions that a program should use when it calls another program. The rest of Part I describes how to use the system services. Part II of this book provides the detailed information for coding the macro instructions.

Summary of Services

The supervisor provides the resources that your programs need while assuring that as many of these resources as possible are being used at a given time. Well designed programs use system resources efficiently. Knowing the conventions and characteristics of the supervisor will help you design more efficient programs.

The services you can request from the supervisor are discussed in the following topics:

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.

Program Management: You can use the supervisor to aid communication between segments of a program. Residency and addressing mode of programs and linkage considerations for MVS/XA are discussed in this section. Save areas, addressability, and passage of control from one segment of a program to another are also discussed.

Resource Control: Portions of some tasks are dependent on the completion of events in other tasks, thus requiring planned task synchronization. Planning is also required when more than one program uses a serially reusable resource.

Program Interruption, Termination, and Dumping Services: 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.

Virtual Storage Management: While virtual storage allows you to write large programs without the need for complex overlay structures, virtual storage must be obtained for your job step. Virtual storage is allocated by both explicit and implicit requests.

Data-in-Virtual: By using a simple technique that lets you create, read, or update external storage data without the traditional GET and PUT access methods, you can write programs that use very large amounts of data. The data, which is not broken up into individual records, appears in your virtual storage all at once. For many applications, this technique also provides better performance than the traditional access methods.

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.

Timing and Communication: The supervisor provides the facilities for timing events and for communication with the system operator, TSO terminals, and the system log.

Linkage Conventions

Linkage conventions are the register and save area conventions a program must follow when it is called by another program or when it calls another program. It is important that all programs follow the linkage conventions described here to ensure that the programs can successfully pass control from one to the other while preserving register contents and parameter data that they need for successful execution.

During the execution of a program the services of another program may be required. The program that requests the services of another program is known as a *calling* program, and the program that was requested is known as the *called* program. For example, when the control program passes control to program A, program A becomes a called program. If program A in turn passes control to program B, program A becomes a calling program, and program B becomes a called program. From the point of view of the control program, however, program A remains a called program until control is returned by program A. For more information on this subject, see the discussion under the heading "Task and Subtask Communications" in "Subtask Creation and Control."

The following conventions are presented assuming one calling and one called program. They apply, however, to all called and calling programs operating in the system. If the conventions presented here are always followed, execution of the called program will not be affected by the method used to pass control or by the identity of the calling program.

Linkage Registers

Registers 0, 1, 13, 14, and 15 are known as the *linkage registers*; they are used in fixed ways by the control program. It is good practice to use these registers in the same way in your program, since they may be modified by the control program or by your program when system macro instructions are used. Registers 2-12 are not changed by the control program.

Registers 0 and 1 are used to pass parameters to the control program or to a called program. The expansions of some system macro instructions result in instructions that load a value into register 0 or 1 or both, or load the address of a parameter list into register 1. For other macro instructions, the control program uses register 1 to pass specified parameters to the program you call.

Register 13 contains the address of the save area provided by the calling program.

Register 14 contains the return address of the calling program or an address within the control program to which your program is to return control when it has completed execution.

Register 15 contains the entry address when control is passed to your program by the control program. The entry address of the called routine should be in register 15 when you pass control to another program. The expansion of some macro instructions results in instructions that load into register 15 the address of a parameter list to be passed to the control program. Register 15 is also used by the called program to return a value (a return code) to the calling program.

The manner in which the control program passes the information in the PARM field of your EXEC statement is a good example of how the control program uses a parameter register to pass information. When control is passed to your program from the control program, register 1 contains the address of a fullword on a fullword boundary in your area of virtual storage (refer to Figure 1). The high-order bit (bit 0) of this word is set to 1. This is a convention used by the control program to indicate the last word in a variable-length parameter list; you must use the same convention when making requests to the control program. Bits 1-31 of the fullword contain the address of a two-byte length field on a halfword boundary. The length field contains a binary count of the number of bytes in the PARM field, which immediately follows the length field. If the PARM field was omitted in the EXEC statement, the count is set to zero. To prevent possible errors, the count should always be used as a length attribute in acquiring the information in the PARM field. If your program is not going to use this information immediately, you should load the address from register 1 into one of registers 2-12 or store the address in a fullword in your program.

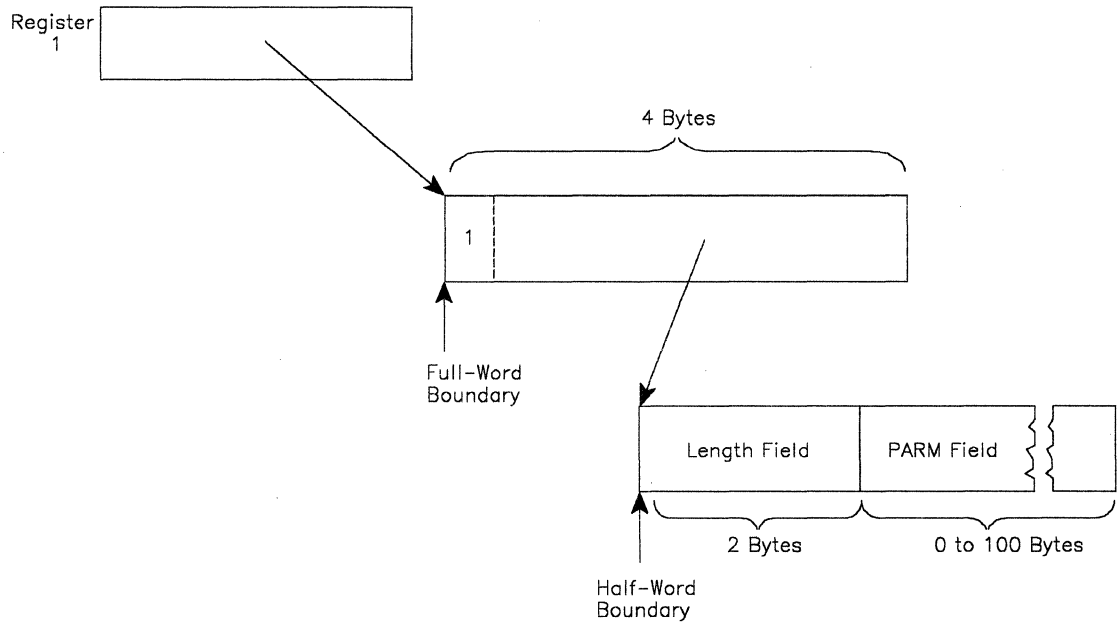

Figure 1. Acquiring PARM Field Information

Saving the Calling Program's Registers

The first action a called program should take is to save the contents of the calling program's registers. The contents of any register the called program modifies and the contents of the linkage registers must be saved. All registers should be saved to avoid errors when the called program is modified.

The registers are saved in the 18-word save area provided by the calling program and pointed to by register 13. The format of this area is shown in Figure 2. As indicated by this figure, the contents of each register must be saved in a specific location within the save area. Registers can be saved either with a store-multiple (STM) instruction or with the SAVE macro instruction. The store-multiple instruction, STM 14,12,12(13), places the contents of all registers except 13 in the proper words of the save area. Saving register 13 is discussed under the heading "Providing a Save Area."

Word	Contents
0	Used by PL/I language program
1	Address of previous save area (stored by calling program)
2	Address of next save area (stored by current program)
3	Register 14 (Return address)
4	Register 15 (Entry address)
5	Register 0
6	Register 1
7	Register 2
8	Register 3
9	Register 4
10	Register 5
11	Register 6
12	Register 7
13	Register 8
14	Register 9
15	Register 10
16	Register 11
17	Register 12

Figure 2. Format of the Save Area

The SAVE macro instruction generates instructions that store a designated group of registers in the save area. The registers to be saved are coded in the same order as in an STM instruction. Figure 3 illustrates uses of the SAVE macro instruction. The T parameter (in B) specifies that the contents of registers 14 and 15 are to be saved.

```
(A) PROGNAME        SAVE (14,12)
(B) PROGNAME        SAVE (5,10),T
```

Figure 3. Use of the SAVE Macro Instruction

The SAVE macro instruction or the equivalent instructions should be placed at the entry point to the program.

Establishing a Base Register

In MVS/XA, addresses are resolved by adding a displacement to a base address. You must, therefore, establish a base register using one of the registers from 2-12 or register 15. If your program does not use system macro instructions and does not pass control to another program, you can establish a base register using the entry address in register 15. Otherwise, because both your program and the control program use register 15 for other purposes, you must establish a base using one of the registers 2-12. This should be done immediately after saving the calling program's registers.

Note: Cautiously choose your base registers keeping in mind that some instructions alter register contents (for example, TRT alters register 2). A complete list of instructions and their processing is available in *Principles of Operation.*

Providing a Save Area

If any control section in your program passes control to another control section, your program must provide its own save area. You must also provide a save area when you use certain system functions, such as GET or PUT. If you establish which registers are available to the called program or control section, a save area is not necessary. Omitting the save area is not a good coding practice, however, since any changes in your program might necessitate changing a called program.

Whether or not your program provides a save area, you must save the address of the calling program's save area, which you used, because you will need it to restore the registers before you return control to the program that called you. If you are not providing a save area, you can keep the address in register 13 or store it in a location in virtual storage. If you are creating your own save area, use the following procedure.

- Store the address of the save area that you used (the address passed to you in register 13) in the second word of the save area you created.
- Store the address of your save area (the address you will pass in register 13) in the third word of the calling program's save area.

This procedure enables you to find the save area when you need it to restore the registers, and it enables a trace from save area to save area should one be necessary during a dump.

Figure 4 and Figure 5 show two methods of obtaining a save area and of saving all the registers, including the addresses of the two save areas. In Figure 4 the registers are stored in the save area provided by the calling program by using the STM instruction. Register 12 is then established as the base register. The address of the caller's save area is then saved in the second word of the new save area, established by the DC statement. The address of the calling program's save area is loaded into register 2. The address of the new save area is loaded into register 13, and then stored in the third word of the caller's save area.

```
PROGNAME    CSECT
PROGNAME    AMODE  31
PROGNAME    RMODE  24
            STM    14,12,12(13)
            LR     12,15
            USING  PROGNAME,12
            ST     13,SAVEAREA+4
            LR     2,13
            LA     13,SAVEAREA
            ST     13,8(2)
            .
            .
            .
SAVEAREA    DC     18F'0'
```

Figure 4. Chaining Save Areas in a Nonreenterable Program

In Figure 5, the SAVE macro instruction is used to store registers. (An STM instruction could have been used.) The entry address is loaded into register 12, which is established as the base register. An unconditional GETMAIN macro instruction (discussed in detail under the heading "Virtual Storage Management") is issued requesting the control program to allocate 72 bytes of virtual storage from an area outside your program, and to return the address of the area in register 1. The addresses of the old and new save areas are stored in the assigned locations, and the address of the new save area is loaded into register 13.

```
PROGNAME    CSECT
PROGNAME    AMODE    31
PROGNAME    RMODE    24
            SAVE     (14,12)
            LR       12,15
            USING    PROGNAME,12
            GETMAIN  R,LV=72
            ST       13,4(1)
            ST       1,8(13)
            LR       13,1
            .
            .
            .
```

Figure 5. Chaining Save Areas in a Reenterable Program

Summary of Conventions to be Followed When Passing and Receiving Control

The following is a list of conventions to be followed when passing and receiving control. The actual methods of passing control are described under the heading "Program Management."

By a calling program before passing control (return required):

- Place the address of your save area in register 13.
- Place the address at which you wish to regain control (the return address) in register 14.
- Place the entry address of the program you are calling in register 15.
- Place the address of the parameter list (if there is one) in register 1. (Passing parameters is discussed under "Program Management.")

By a calling program before passing control (no return required):

- Restore registers 2-12 and 14.
- Place the address of the save area provided by the program that called you in register 13.
- Place the entry address of the program you are calling in register 15.
- Place the addresses of parameter lists in registers 1 and 0.

By a called program upon receiving control:

- Save the contents of registers 0-12, 14, and 15 in the save area provided by the calling program.
- Establish a base register.
- Request the control program to allocate storage for a save area if you did not already allocate it by using a DC statement.
- Store the save area addresses in the assigned locations.

By a called program before returning control:

- Restore registers 0-12 and 14.
- Place the address of the save area provided by the program you are returning control to in register 13.
- Place a return code in register 15 if one is required. Otherwise, restore register 15.

Subtask Creation and Control

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

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

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

Creating the Task

A new task is created by issuing an ATTACH macro instruction. The task that is active when the ATTACH macro instruction is issued is the originating task; the newly created task is the subtask of the originating task. The subtask competes for control in the same manner as any other task in the system, on the basis of priority (both address space priority and task priority within the address space) and the current ability to use a processor. The address of the task control block for the subtask is returned in register 1.

If the ATTACH macro instruction is executed successfully, control is returned to the user with a return code of X'00' in register 15.

The entry point in the load module to be given control when the subtask becomes active is specified as it is in a LINK macro instruction, that is, through the use of the EP, EPLOC, and DE parameters. The use of these parameters is discussed in "Program Management." Parameters can be passed to the subtask using the PARAM and VL parameters, also described under "The LINK Macro Instruction." Additional parameters deal with the priority of the subtask, provide for communication between tasks, specify libraries to be used for program linkages, and establish an error recovery environment for the new subtask.

Priorities

There are really three priorities to consider: address space priorities, task priorities, and subtask priorities.

Address Space Priority

When each job is initiated, an address space is created. All successive steps in the job execute in the same address space. The address space has a dispatching priority, which is normally determined by the control program. The control program will select, and alter, the priority in order to achieve the best load balance in the system - that is, in order to make the most efficient use of processor time and other system resources.

It may be desirable for some jobs to execute at a different address space priority than the default priority assigned by the system. To assign a priority, you code DPRTY=(value1,value2) on the EXEC statement. The address space priority is then determined as follows:

address space dispatching priority = (value1 x 16) + value2

Once the address space dispatching priority is set, it can be altered only by the control program. (Thus, there is no limit priority associated with an address space.) However, a new address space priority may be set for succeeding job steps by specifying different values in the DPRTY parameter on the EXEC statement.

The IEAIPSxx and IEAICSxx members of SYS1.PARMLIB can override the dispatching priority specified by the DPRTY parameter. The control program assigns the priority obtained from IEAIPSxx to jobsteps that request a dispatching priority falling within specific installation defined limits. IEAICSxx directs jobs into specific performance groups thereby affecting their priority. See *SPL: Initialization and Tuning* for additional information.

Task Priority

Each task in an address space has associated with it a limit priority and a dispatching priority. These priorities are set by the control program when a job step is initiated. When other tasks are created in the address space by use of the ATTACH macro instruction, they may be given different limit and dispatching priorities by use of the LPMOD and DPMOD parameters, respectively.

The task dispatching priorities of the tasks in an address space do not affect the order in which the jobs are selected for execution because the order is selected on the basis of address space dispatching priority. Once an address space is selected for dispatching, the highest priority task awaiting execution is selected. Thus, task priorities may affect processing within an address space. Note, however, that in a multiprocessing system, task priorities cannot guarantee the order in which the tasks will execute because more than one task may be executing simultaneously in the same address space on different processors. In a paging environment, page faults may alter the order in which the tasks execute.

Subtask Priority

When a subtask is created, the limit and dispatching priorities of the subtask are the same as the current limit and dispatching priorities of the originating task except when the subtask priorities are modified by the LPMOD and DPMOD parameters of the ATTACH macro instruction. The LPMOD parameter specifies the number to be subtracted from the current limit priority of the originating task. The result of the subtraction is assigned as the limit priority of the subtask. If the result is zero or negative, zero is assigned as the limit priority. The DPMOD parameter specifies the number to be added to the current dispatching priority of the originating task. The result of the addition is assigned as the dispatching priority of the subtask, unless the number is greater than the limit priority or less than zero. In that case, the limit priority or 0, respectively, is used as the dispatching priority.

Assigning and Changing Priority

Tasks with a large number of input/output operations should be assigned a higher priority than tasks with little input/output, because the tasks with much input/output will be in a wait condition for a greater amount of time. The lower priority tasks will be executed when the higher priority tasks are in a wait condition. As the input/output operations are completed, the higher priority tasks get control, so that more I/O can be started.

The priorities of subtasks can be changed by using the CHAP macro instruction. The CHAP macro instruction changes the dispatching priority of the active task or one of its subtasks by adding a positive or negative value. The dispatching priority of an active task can be made less than the dispatching priority of another task. If this occurs, if the other task is dispatchable it would be given control after execution of the CHAP macro instruction.

The CHAP macro instruction can also be used to increase the limit priority of any of an active task's subtasks. An active task cannot change its own limit priority. The dispatching priority of a subtask can be raised above its own limit priority, but not above the limit of the originating task. When the dispatching priority of a subtask is raised above its own limit priority, the subtask's limit priority is automatically raised to equal its new dispatching priority.

Task and Subtask Communications

The task management information in this section is required only for establishing communications among tasks in the same job step. The relationship of tasks in a job step is shown in Figure 6. The horizontal lines in Figure 6 separate originating tasks and subtasks; they have no bearing on task priority. Tasks A, A1, A2, A2a, B, B1 and B1a are all subtasks of the job-step task; tasks A1, A2, and A2a are subtasks of task A. Tasks A2a and B1a are the lowest level tasks in the job step. Although task B1 is at the same level as tasks A1 and A2, it is not considered a subtask of task A.

Task A is the originating task for both tasks A1 and A2, and task A2 is the originating task for task A2a. A hierarchy of tasks exists within the job step. Therefore the job step task, task A, and task A2 are predecessors of task A2a, while task B has no direct relationship to task A2a.

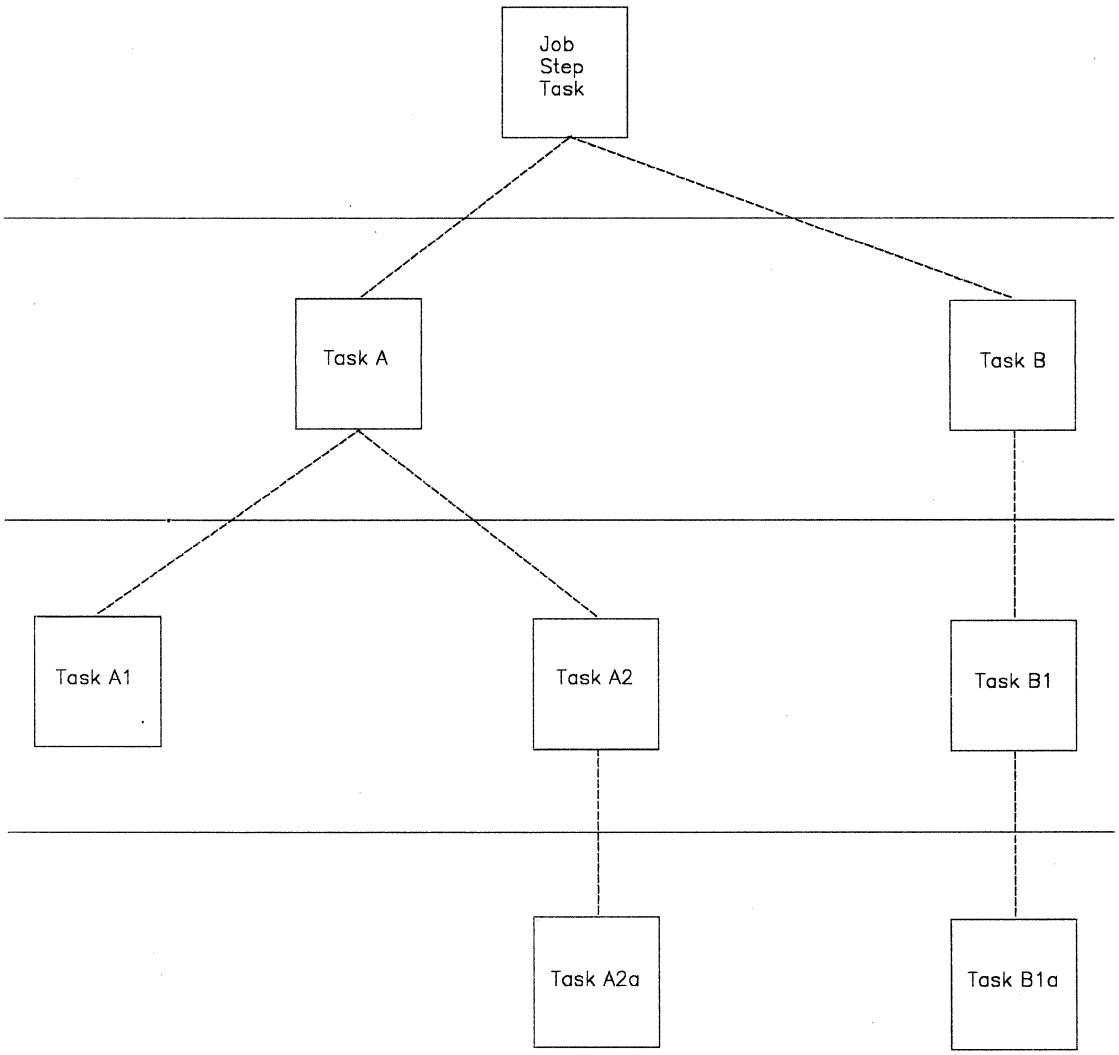

Figure 6. Levels of Tasks in a Job Step

All of the tasks in the job step compete independently for processor time; if no constraints are provided, the tasks are performed and are terminated asynchronously. However, since each task is performing a portion of the same job step, some communication and constraints between tasks are required, such as notification of the completion of subtasks. If termination of a predecessor task is attempted before all of the subtasks are complete, those subtasks and the predecessor task are abnormally terminated.

Two parameters, the ECB and ETXR parameters, are provided in the ATTACH macro instruction to assist in communication between a subtask and the originating task. These parameters are used to indicate the normal or abnormal termination of a subtask to the originating task. If the ECB or ETXR parameter, or both, are coded in the ATTACH macro instruction, the task control block of the subtask is not removed from the system when the subtask is terminated. The originating task must remove the task control block from the system after termination of the subtask by issuing a DETACH macro instruction. If the ECB parameter is specified in the ATTACH macro instruction, the ECB must be in storage so that the issuer of the attach can wait on it (using the WAIT macro instruction) and the control program can post it on behalf of the terminating task. The task control blocks for all subtasks must be removed before the originating task can terminate normally.

The ETXR parameter specifies the address of an end-of-task exit routine in the originating task, which is to be given control when the subtask being created is terminated. The end-of-task routine is given control asynchronously after the subtask has terminated and must therefore be in virtual storage when it is required. After the control program terminates the subtask, the end-of-task routine specified is scheduled to be executed. It competes for CPU time using the priority of the originating task and of its address space and can be given control even though the originating task is in the wait condition. Although the DETACH macro instruction does not have to be issued in the end-of-task routine, this is a good place for it.

The ECB parameter specifies the address of an event control block (discussed under "Task Synchronization"), which is posted by the control program when the subtask is terminated. After posting occurs, the event control block contains the completion code specified for the subtask.

If neither the ECB nor the ETXR parameter is specified in the ATTACH macro instruction, the task control block for the subtask is removed from the system when the subtask is terminated. Its originating task does not have to issue a DETACH macro instruction. A reference to the task control block in a CHAP or a DETACH macro instruction in this case is risky as is task termination. Since the originating task is not notified of subtask termination, you may refer to a task control block that has been removed from the system, which would cause the active task to be abnormally terminated.

Program Management

This section discusses facilities that will help you to design your programs. It includes descriptions of the residency mode and addressing mode of programs, linkage considerations for MVS/XA, load module structures, facilities for passing control between programs, and the use of the associated macro instructions.

Residency and Addressing Mode of Programs

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

When a program is executing in 24-bit addressing mode, the system treats both instruction and data addresses as 24-bit addresses. This allows programs executing in 24-bit addressing mode to address 16 megabytes (16,777,216 bytes) of storage. Similarly, when a program is executing in 31-bit addressing mode, the system treats both instruction and data addresses as 31-bit addresses. This allows a program executing in 31-bit addressing mode to address 2 gigabytes (2,147,483,648 bytes or 128 x 16 megabytes) of storage.

You can define the residency mode and the addressing mode of a program in the source code. Figure 7 shows an example of the definition of the AMODE and RMODE attributes in the source code. This example defines the addressing mode of the load module as 31-bit and the residency mode of the load module as 24-bit. Therefore, the program will receive control in 31-bit addressing mode and will reside below 16 Mb.

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

Figure 7. Assembler Definition of AMODE/RMODE

Version 2 of Assembler H places the AMODE and RMODE in the external symbol dictionary (ESD) of the output object module for use by the linkage editor. The linkage editor passes this information on to the control program through the directory entry for the partitioned data set (PDS) that contains the load module and the composite external symbol dictionary (CESD) record in the load module. You can also specify the AMODE/RMODE attributes of a load module by using linkage editor control cards. *SPL: 31-bit Addressing* contains additional information about residency and addressing mode; *Linkage Editor and Loader* contains information about the linkage editor control cards.

Residency Mode Definitions

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

24 specifies that the program must reside in 24-bit addressable virtual storage.

ANY specifies that the program can reside anywhere in virtual storage because the code has no virtual storage residency restrictions.

Note: The default value for RMODE is 24.

Addressing Mode Definitions

The AMODE attribute, located in the PDS directory entry for the module, specifies the addressing mode that the module expects at entry. Bit 32 of the program status word (PSW) indicates the addressing mode of the program that is executing. 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.

Note: The default value for AMODE is 24.

Linkage Considerations for MVS/XA

MVS/XA supports programs that execute in either 24-bit or 31-bit addressing mode. The following branch instructions take addressing mode into consideration:

Branch and link (BAL)
Branch and link, register form (BALR)
Branch and save (BAS)
Branch and save, register form (BASR)
Branch and set mode (BSM)
Branch and save and set mode (BASSM)

See *Principles of Operation* for a complete description of how these instructions function. The following paragraphs provide a general description of these branch instructions in MVS/XA.

The BAL and BALR instructions are unconditional branch instructions (to the address in operand 2). BAL and BALR function differently depending on the addressing mode in which you are executing. The difference is in the linkage information passed in the link register when these instructions execute. In 31-bit addressing mode, the link register contains the AMODE indicator (bit 0) and the address of the next sequential instruction (bits 1-31); in 24-bit addressing mode, the link register contains the instruction length code, condition code, program mask, and the address of the next sequential instruction.

BAS and BASR perform the same function that BAL and BALR perform when BAL and BALR execute in 31-bit addressing mode.

The BSM instruction provides problem programs with a way to change the AMODE bit in the PSW. BSM is an unconditional branch instruction (to the address in operand 2) that saves the current AMODE in the high-order bit of the link register (operand 1), and sets the AMODE indicator in the PSW to agree with the AMODE of the address to which you are transferring control (that is, the high order bit of operand 2).

The BASSM instruction functions in a manner similar to the BSM instruction. In addition to saving the current AMODE in the link register, setting the PSW AMODE bit, and transferring control, BASSM also saves the address of the next sequential instruction in the link register thereby providing a return address.

BASSM and BSM are used for entry and return linkage in a manner similar to BALR and BR. The major difference from BALR and BR is that BASSM and BSM can save and change addressing mode.

Passing Control Between Programs with the Same AMODE

If you are passing control between programs that execute in the same addressing mode, there are several combinations of instructions that you can use. Some of these combinations are:

Transfer	Return
BAL/BALR	BR
BAS/BASR	BR

Passing Control Between Programs with Different AMODEs

If you are passing control between programs executing in different addressing modes, the AMODE indicator in the PSW must be changed. The BASSM and BSM instructions perform this function for you. You can transfer to a program in another AMODE using a BASSM instruction and then return by means of a BSM instruction. This sequence of instructions ensures that both programs execute in the correct AMODE.

Figure 8 shows an example of passing control between programs with different addressing modes. In the example, TEST executes in 24-bit AMODE and EP1 executes in 31-bit AMODE. Before transferring control to EP1, the TEST program loads register 15 with EPA, the pointer defined entry point address (that is, the address of EP1 with the high order bit set to 1 to indicate 31-bit AMODE). This is followed by a BASSM 14,15 instruction, which performs the following functions:

- Sets the high-order bit of the link register (register 14) to 0 (because TEST is currently executing in 24-bit AMODE) and puts the address of the next sequential instruction into bits 1-31.
- Sets the PSW AMODE bit to 1 to agree with bit 0 of register 15.
- Transfers to EP1 (the address in bits 1-31 of register 15).

The EP1 program executes in 31-bit AMODE. Upon completion, EP1 sets a return code in register 15 and executes a BSM 0,14 instruction, which performs the following functions:

- Sets the PSW AMODE bit to 0 to correspond to the high-order bit of register 14.
- Transfers control to the address following the BASSM instruction in the TEST program.

```
TEST    CSECT
TEST    AMODE   24
TEST    RMODE   24
        .
        .
        .
        L       15,EPA      OBTAIN TRANSFER ADDRESS
        BASSM   14,15       SWITCH AMODE AND TRANSFER
        .
        .
        .
        EXTRN   EP1
EPA     DC      A(X'80000000'+EP1) POINTER DEFINED ENTRY POINT ADDRESS
        .
        .
        .
        END
```

```
EP1     CSECT
EP1     AMODE   31
EP1     RMODE   ANY
        .
        .
        .
        SLR     15,15       SET RETURN CODE 0
        BSM     0,14        RETURN TO CALLER'S AMODE AND TRANSFER
        END
```

Figure 8. Example of Addressing Mode Switch

Load Module Structure Types

Each load module used during a job step can be designed in one of three load module structures: *simple*, *planned overlay*, or *dynamic*. A simple structure does not pass control to any other load modules during its execution, and is brought into virtual storage all at one time. A planned overlay structure may, if necessary, pass control to other load modules during its execution, and it is not brought into virtual storage all at one time. Instead, segments of the load module reuse the same area of virtual storage. A dynamic structure is brought into virtual storage all at one time, and passes control to other load modules during its execution. Each of the load modules to which control is passed can be one of the three structure types. Characteristics of the load module structure types are summarized in Figure 9.

Since the large capacity of virtual storage all but eliminates the need for complex overlay structures, planned overlays will not be discussed further.

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

Figure 9. Characteristics of Load Modules

Simple Structure

A simple structure consists of a single load module produced by the linkage editor. The single load module contains all of the instructions required and is paged into real storage by the control program as it is executed. The simple structure can be the most efficient of the two structure types because the instructions it uses to pass control do not require control-program assistance. However, you should design your program to make most efficient use of paging.

Dynamic Structure

A dynamic structure requires more than one load module during execution. Each load module required can operate as either a simple structure or another dynamic structure. The advantages of a dynamic structure over a simple structure increase as the program becomes more complex, particularly when the logical path of the program depends on the data being processed. The load modules required in a dynamic structure are paged into real storage when required, and can be deleted from virtual storage when their use is completed.

Load Module Execution

Depending on the configuration of the operating system and the macro instructions used to pass control, execution of the load modules is serial or in parallel. Execution is serial in the MVS/XA operating system unless an ATTACH macro instruction is used to create a new task. The new task competes for processor time independently with all other tasks in the system. The load module named in the ATTACH macro instruction is executed in parallel with the load module containing the ATTACH macro instruction. The execution of the load modules is serial within each task.

The following paragraphs discuss passing control for serial execution of a load module. Creation and management of new tasks is discussed under the headings "Task Creation and Control."

Passing Control in a Simple Structure

There are certain procedures to follow when passing control to an entry point in the same load module. The established conventions to use when passing control are also discussed. These procedures and conventions are the framework for all program interfaces. Knowledge of the information about addressing contained in the *Assembler Language* publication is required.

Passing Control without Return

Some control sections pass control to another control section of the load module and do not receive control back. An example of this type of control section is a housekeeping routine at the beginning of a program that establishes values, initializes switches, and acquires buffers for the other control sections in the program. Use the following procedures when passing control without return.

Preparing to Pass Control

Because control will not be returned to this control section, you must restore the contents of register 14. Register 14 originally contained the address of the location in the calling program (for example, the control program) to which control is to be passed when your program is finished. Since the current control section does not make the return to the calling program, the return address must be passed on to the control section that does make the return. In addition, the contents of registers 2-12 must be unchanged when your program eventually returns control, so these registers must also be restored.

If control were being passed to the next entry point from the control program, register 15 would contain the entry address. You should use register 15 in the same way, so that the called routine remains independent of the program that passed control to it.

Use register 1 to pass parameters. Establish a parameter list and place the address of the list in register 1. The parameter list should consist of consecutive fullwords starting on a fullword boundary, each fullword containing an *address* to be passed to the called control section. When executing in 24-bit AMODE, each address is located in the three low-order bytes of the word. When executing in 31-bit AMODE, each address is located in bits 1-31 the word. In both addressing modes, set the high-order bit of the last word to 1 to indicate that it is the last word of the list. The system convention is that the list contain addresses only. You may, of course, deviate from this convention; however, when you deviate from any system convention, you restrict the use of your programs to those programmers who know about your special conventions.

Since you have reloaded all the necessary registers, the save area that you received on entry is now available, and should be reused by the called control section. Pass the address of the save area in register 13 just as it was passed to you. By passing the address of the old save area, you save the 72 bytes of virtual storage for a second, unnecessary, save area.

Note: If you pass a new save area instead of the one received on entry, errors could occur.

Passing Control

The common way to pass control between one control section and an entry point in the same load module is to load register 15 with a V-type address constant for the name of the external entry point, and then to branch to the address in register 15. The external entry point must have been identified using an ENTRY instruction in the called control section if the entry point is not the same as the control section's CSECT name.

Figure 10 shows an example of loading registers and passing control. In this example, no new save area is used, so register 13 still contains the address of the old save area. It is also assumed for this example that the control section will pass the same parameters it received to the next entry point. First, register 14 is reloaded with the return address. Next, register 15 is loaded with the address of the external entry point NEXT, using the V-type address constant at the location NEXTADDR. Registers 0-12 are reloaded, and control is passed by a branch instruction using register 15. The control section to which control is passed contains an ENTRY instruction identifying the entry point NEXT.

```
            .
            .
            .
            L    14,12(13)        LOAD CALLER'S RETURN ADDRESS
            L    15,NEXTADDR      ENTRY NEXT
            LM   0,12,20(13)      RETURN CALLER's REGISTERS
            BR   15               NEXT SAVE (14,12)
            .
            .
            .
NEXTADDR    DC   V(NEXT)
```

Figure 10. Passing Control in a Simple Structure

Figure 11 shows an example of passing a parameter list to an entry point with the same addressing mode. Early in the routine the contents of register 1 (that is, the address of the fullword containing the PARM field address) were stored at the fullword PARMADDR. Register 13 is loaded with the address of the old save area, which had been saved in word 2 of the new save area. The contents of register 14 are restored, and register 15 is loaded with the entry address.

```
         .
         .
         .
         USING *,12            Establish addressability
EARLY    ST    1,PARMADDR      Save parameter address
         .
         .
         .
         L     13,4(13)        Reload address of old save area
         L     0,20(13)
         L     14,12(13)       Load return address
         L     15,NEXTADDR     Load address of next entry point
         LA    1,PARMLIST      Load address of parameter list
         OI    PARMADDR,X'80'  Turn on last parameter indicator
         LM    2,12,28(13)     Reload remaining registers
         BR    15              Pass control
         .
         .
         .
PARMLIST DS    0A
DCBADDRS DC    A(INDCB)
         DC    A(OUTDCB)
PARMADDR DC    A(0)
NEXTADDR DC    V(NEXT)
```

Figure 11. Passing Control With a Parameter List

The address of the list of parameters is loaded into register 1. These parameters include the addresses of two data control blocks (DCBs) and the original register 1 contents. The high-order bit in the last address parameter (PARMADDR) is set to 1 using an OR-immediate instruction. The contents of registers 2-12 are restored. (Because one of these registers was the base register, restoring the registers earlier would have made the parameter list unaddressable.) A branch register instruction using register 15 passes control to entry point NEXT.

Passing Control with Return

The control program passed control to your program, and your program will return control when it is through processing. Similarly, control sections within your program will pass control to other control sections, and expect to receive control back. An example of this type of control section is a monitoring routine; the monitor determines the order of execution of other control sections based on the type of input data. Use the following procedures when passing control with return.

Preparing to Pass Control

Use registers 15 and 1 in the same manner they are used to pass control without return. Register 15 contains the entry address in the new control section and register 1 is used to pass a parameter list.

Register 14 must contain the address of the location to which control is to be returned when the called control section completes execution. The address can be the instruction following the instruction which causes control to pass, or it can be another location within the current control section designed to handle all returns. Registers 2-12 are not involved in the passing of control; the called control section should not depend on the contents of these registers in any way.

You should provide a new save area for use by the called control section as previously described, and pass the address of that save area in register 13. Note that the same save area can be reused after control is returned by the called control section. One new save area is ordinarily all you will require regardless of the number of control sections called.

Passing Control

Two standard methods are used for passing control to another control section and providing for return of control. One is an extension of the method used to pass control without a return, and requires a V-type address constant and a branch, a branch and link, or a branch and save instruction provided both programs execute in the same addressing mode. If the addressing mode changes, a branch and save and set mode instruction should be used. The other method uses the CALL macro instruction to provide a parameter list and establish the entry and return addresses. With either method, you must identify the entry point by an ENTRY instruction in the called control section if the entry name is not the same as the control section CSECT name. Figure 12 and Figure 13 illustrate the two methods of passing control; in each example, assume that register 13 already contains the address of a new save area.

Figure 12 also shows the use of an inline parameter list and an answer area. The address of the external entry point is loaded into register 15 in the usual manner. A branch and link instruction is then used to branch around the parameter list and to load register 1 with the address of the parameter list. An inline parameter list, such as the one shown in Figure 12, is convenient when you are debugging because the parameters involved are located in the listing (or the dump) at the point they are used, instead of at the end of the listing or dump. Note that the high-order bit of the last address parameter (ANSWERAD) is set to 1 to indicate the end of the list. The area pointed to by the address in the ANSWERAD parameter is an area to be used by the called control section to pass parameters back to the calling control section. This is a possible method to use when a called control section must pass parameters back to the calling control section. Parameters are passed back in this manner so that no additional registers are involved. The area used in this example is twelve words. The size of the area for any specific application depends on the requirements of the two control sections involved.

```
            .
            .
            .
            L       15,NEXTADDR           Entry address in register 15
            CNOP    0,4
            BAL     1,GOOUT               Parameter list address in
                                          register 1
PARMLIST    DS      0A                    Start of parameter list
DCBADDRS    DC      A(INDCB)              Input DCB address
            DC      A(OUTDCB)             Output DCB address
ANSWERAD    DC      A(AREA+X'80000000')   Answer area address with
                                          high-order bit on
NEXTADDR    DC      V(NEXT)               Address of entry point
GOOUT       BALR    14,15                 Pass control; register 14
                                          contains return address
                                          and current AMODE
RETURNPT    ...
AREA        DC      12F'0'                Answer area from NEXT
```

Note: This example assumes that you are passing control to a program that executes in the same addressing mode as your program. See the topic "Linkage Considerations for MVS/XA" for information on how to handle branches between programs that execute in different addressing modes.

Figure 12. Passing Control With Return

```
            CALL    NEXT,(INDCB,OUTDCB,AREA),VL
RETURNPT    ...
AREA        DC      12F'0'
```

Note: You cannot use the CALL macro instruction to pass control to a program that executes in a different addressing mode.

Figure 13. Passing Control With CALL

The CALL macro instruction in Figure 13 provides the same functions as the instructions in Figure 12. When the CALL macro instruction is expanded, the parameters cause the following results:

NEXT - A V-type address constant is created for NEXT, and the address is loaded into register 15.

(INDCB,OUTDCB,AREA) - A-type address constants are created for the three parameters coded within parentheses, and the address of the first A-type address constant is placed in register 1.

VL - The high-order bit of the last A-type address constant is set to 1.

Control is passed to NEXT using a branch and link instruction. The address of the instruction following the CALL macro instruction is loaded into register 14 before control is passed.

In addition to the results described above, the V-type address constant generated by the CALL macro instruction requires the load module with the entry point NEXT to be link edited into the same load module as the control section containing the CALL macro instruction. The *Linkage Editor and Loader* publication tells more about this service.

The parameter list constructed from the CALL macro instruction in Figure 13, contains only A-type address constants. A variation on this type of parameter list results from the following coding:

```
CALL    NEXT,(INDCB,(6),(7)),VL
```

In the above CALL macro instruction, two of the parameters to be passed are coded as registers rather than symbolic addresses. The expansion of this macro instruction again results in a three-word parameter list; in this example, however, the expansion also contains instructions that store the contents of registers 6 and 7 in the second and third words, respectively, of the parameter list. The high-order bit in the third word is set to 1 after register 7 is stored. You can specify as many address parameters as you need, and you can use symbolic addresses or register contents as you see fit.

Analyzing the Return

When the control program returns control to the user after it invokes a system service, the contents of registers 2-13 are unchanged. When control is returned to your control section from the called control section, registers 2-14 contain the same information they contained when control was passed, as long as system conventions are followed. The called control section has no obligation to restore registers 0 and 1; so the contents of these registers may or may not have been changed.

When control is returned, register 15 can contain a return code indicating the results of the processing done by the called control section. If used, the return code should be a multiple of four, so a branching table can be used easily, and a return code of zero should be used to indicate a normal return. The control program frequently uses this method to indicate the results of the requests you make using system macro instructions; an example of the type of return codes the control program provides is shown in the description of the IDENTIFY macro instruction.

The meaning of each of the codes to be returned must be agreed upon in advance. In some cases, either a "good" or "bad" indication (zero or nonzero) will be sufficient for you to decide your next action. If this is true, the coding in Figure 14 could be used to analyze the results. Many times, however, the results and the alternatives are more complicated, and a branching table, such as shown in Figure 15 could be used to pass control to the proper routine.

Note: Explicit tests are required to ensure that the return code value does not exceed the branch table size.

```
RETURNPT    LTR     15,15     Test return code for zero
            BNZ     ERRORTN   Branch if not zero to error
                              routine
            .
            .
            .
```

Figure 14. Test for Normal Return

```
RETURNPT    B    RETTAB(15)   Branch to table using return
                               code
RETTAB      B    NORMAL       Branch to normal routine
            B    COND1        Branch to routine for
                              condition 1
            B    COND2        Branch to routine for
                              condition 2
            B    GIVEUP       Branch to routine to handle
                              impossible situations
            .
            .
            .
```

Figure 15. Return Code Test Using Branching Table

How Control is Returned

In the discussion of the return under "Analyzing the Return" it was indicated that the control section returning control must restore the contents of registers 2-14. Because these are the same registers reloaded when control is passed without a return, refer to the discussion under "Passing Control without Return" for detailed information and examples. The contents of registers 0 and 1 do not have to be restored.

Register 15 can contain a return code when control is returned. As indicated previously, a return code should be a multiple of four with a return code of zero indicating a normal return. The return codes other than zero that you use can have any meaning, as long as the control section receiving the return codes is aware of that meaning.

The return address is the address originally passed in register 14; you should always return control to that address. If an addressing mode switch is not involved, you can either use a branch instruction such as BR 14, or you can use the RETURN macro instruction. An example of each of these methods of returning control is discussed in the following paragraphs. If an addressing mode switch is involved, you can use a BSM 0,14 instruction to return control. See Figure 8 for an example that uses the BSM instruction to return control.

Figure 16 shows a portion of a control section used to analyze input data cards and to check for an out-of-tolerance condition. Each time an out-of-tolerance condition is found, in addition to some corrective action, one is added to the one-byte value at the address STATUSBY. After the last data card is analyzed, this control section returns to the calling control section, which bases its next action on the number of out-of-tolerance conditions encountered. The coding shown in Figure 16 loads register 14 with the return address. The contents of register 15 are set to zero, and the value at the address STATUSBY (the number of errors) is placed in the low-order eight bits of the register. The contents of register 15 are shifted to the left two places to make the value a multiple of four. Registers 2-12 are reloaded, and control is returned to the address in register 14.

```
          .
          .
          .
          L      13,4(13)        Load address of previous save
                                 area
          L      14,12(13)       Load return address
          SR     15,15           Set register 15 to zero
          IC     15,STATUSBY     Load number of errors
          SLA    15,2            Set return code to multiple
                                 of 4
          LM     2,12,28(13)     Reload registers 2-12
          BR     14              Return
          .
          .
          .
STATUSBY  DC     X'00'
```

Note: This example assumes that you are returning to a program with the same AMODE. If not, use the BSM instruction to transfer control.

Figure 16. Establishing a Return Code

The RETURN macro instruction saves coding time. The expansion of the RETURN macro instruction provides instructions that restore a designated range of registers, load a return code in register 15, and branch to the address in register 14. If T is specified, the RETURN macro instruction flags the save area used by the returning control section (that is, the save area supplied by the calling routine). It does this by setting the low-order bit of word four of the save area to one after the registers have been restored. The flag indicates that the control section that used the save area has returned to the calling control section. The flag is useful when tracing the flow of your program in a dump. For a complete record of program flow, a separate save area must be provided by each control section each time control is passed.

You must restore the contents of register 13 before issuing the RETURN macro instruction. Code the registers to be reloaded in the same order as they would have been designated for a load-multiple (LM) instruction. You can load register 15 with the return code before you write the RETURN macro instruction, you can specify the return code in the RETURN macro instruction, or you can reload register 15 from the save area.

The coding shown in Figure 17 provides the same result as the coding shown in Figure 16. Registers 13 and 14 are reloaded, and the return code is loaded in register 15. The RETURN macro instruction reloads registers 2-12 and passes control to the address in register 14. The save area used is not flagged. The RC=(15) parameter indicates that register 15 already contains the return code, and the contents of register 15 are not to be altered.

```
          .
          .
          .
          L         13,4(13)          Restore save area address
          L         14,12(13)         Return address in
                                      register 14
          SR        15,15             Zero register 15
          IC        15,STATUSBY       Load number of errors
          SLA       15,2              Set return code to
                                      multiple of 4
          RETURN    (2,12),RC=(15)    Reload registers and
                                      return
          .
          .
          .
STATUSBY  DC        X'00'
```

Note: You cannot use the RETURN macro instruction to pass control to a program that executes in a different addressing mode.

Figure 17. Using the RETURN Macro Instruction

Figure 18 illustrates another use of the RETURN macro instruction. The correct save area address is again established, and then the RETURN macro instruction is issued. In this example, registers 14 and 0-12 are reloaded, a return code of 8 is placed in register 15, the save area is flagged, and control is returned. Specifying a return code overrides the request to restore register 15 even though register 15 is within the designated range of registers.

```
          .
          .
          .
          L         13,4(13)
          RETURN    (14,12),T,RC=8
```

Figure 18. RETURN Macro Instruction With Flag

Return to the Control Program

The discussion in the preceding paragraphs has covered passing control within one load module, and has been based on the assumption that the load module was brought into virtual storage because of the program name specified in the EXEC statement. The control program established only one task to be performed for the job step. When the logical end of the program is reached, control passes to the return address passed (in register 14) to the first control section in the control program. When the control program receives control at this point, it terminates the task it created for the job step, compares the return code in register 15 with any COND values specified on the JOB and EXEC statements, and determines whether or not subsequent job steps, if any are present, should be executed.

Passing Control in a Dynamic Structure

The discussion of passing control in a simple structure provides the background for the discussion of passing control in a dynamic structure. Within each load module, control should be passed as in a simple structure. If you can determine which control sections will make up a load module before you code the control sections, you should pass control within the load module without involving the control program. The macro instructions discussed in this section provide increased linkage capability, but they require control program assistance and possibly increased execution time.

Bringing the Load Module into Virtual Storage

The load module containing the entry name you specified on the EXEC statement is automatically brought into virtual storage by the control program. The control program places the load module above or below 16 Mb according to its RMODE attribute. Any other load modules you require during your job step are brought into virtual storage by the control program when requested; these requests are made by using the LOAD, LINK, ATTACH, and XCTL macro instructions. The LOAD macro instruction sets the high-order bit of the entry point address to indicate the addressing mode of the load module. The ATTACH, LINK, and XCTL macro instructions use this information to set the addressing mode for the module that gets control. If the AMODE is ANY, the module will get control in the same addressing mode as the program that issued the ATTACH, LINK, or XCTL macro instruction. If a copy of the load module must be brought into storage, the control program places the load module above or below 16 Mb according to its RMODE attribute. The following paragraphs discuss the proper use of these macro instructions.

Location of the Load Module

Initially, each load module that you can obtain dynamically is located in a library (partitioned data set). This library is the link library, the job or step library, the task library, or a private library.

- The link library is always present and is available to all job steps of all jobs. The control program provides the data control block for the library and logically connects the library to your program, making the members of the library available to your program.

- The job and step libraries are explicitly established by including //JOBLIB and //STEPLIB DD statements in the input stream. The //JOBLIB DD statement is placed immediately after the JOB statement, while the //STEPLIB DD statement is placed among the DD statements for a particular job step. The job library is available to all steps of your job, except those that have step libraries. A step library is available to a single job step; if there is a job library, the step library replaces the job library for the step. For either the job library or the step library, the control program provides the data control block and issues the OPEN macro instruction to logically connect the library to your program.

- Unique task libraries can be established by using the TASKLIB parameter of the ATTACH macro instruction. The issuer of the ATTACH macro instruction is responsible for providing the DD statement and opening the data set or sets. If the TASKLIB parameter is omitted, the task library of the attaching task is propagated to the attached task. In the following example, task A's job library is LIB1. Task A attaches task B, specifying TASKLIB=LIB2 in the ATTACH macro instruction. Task B's task library is therefore LIB2. When task B attaches task C, LIB2 is searched for task C before LIB1 or the link library. Because task B did not specify a unique task library for task C, its own task library (LIB2) is propagated to task C and is the first library searched when task C requests that a module be brought into virtual storage.

```
Task A    ATTACH EP=B,TASKLIB=LIB2
Task B    ATTACH EP=C
```

- A private library is defined by including a DD statement in the input stream and is available only to the job step in which it is defined. You must provide the data control block and issue the OPEN macro instruction for each data set. You may use more than one private library by including more than one DD statement and an associated data control block.

A library can be a single partitioned data set, or a collection of such data sets. When it is a collection, you define each data set by a separate DD statement, but you assign a name only to the statement that defines the first data set. Thus, a job library consisting of three partitioned data sets would be defined as follows:

```
//JOBLIB    DD  DSNAME=PDS1,...
//          DD  DSNAME=PDS2,...
//          DD  DSNAME=PDS3...
```

The three data sets (PDS1, PDS2, PDS3) are processed as one, and are said to be *concatenated.* Concatenation and the use of partitioned data sets is discussed in more detail in the *Data Management Services* publication.

Some of the load modules from the link library may already be in virtual storage in an area called the link pack area. The contents of these areas are determined during the nucleus initialization process and will vary depending on the requirements of your installation. The link pack area contains all reenterable load modules from the LPA library, along with installation selected modules from the SVC and link libraries. These load modules can be used by any job step in any job.

With the exception of those load modules contained in this area, copies of all of the reenterable load modules you request are brought into your area of virtual storage and are available to any task in your job step. The portion of your area containing the copies of the load modules is called the job pack area.

The Search for the Load Module

In response to your request for a copy of a load module, the control program searches the job pack area, the task's load list, and the link pack area. If a copy of the load module is found in one of the pack areas, the control program determines whether that copy can be used (see "Using an Existing Copy"). If an existing copy can be used, the search stops. If it cannot be used, the search continues until the module is located in a library. The load module is then brought into the job pack area or the load list area.

The order in which the libraries and pack areas are searched depends on the parameters used in the macro instruction (LINK, LOAD, XCTL, or ATTACH) requesting the load module. The parameters that define the order of the search are EP, EPLOC, DE, DCB, and TASKLIB.

The TASKLIB parameter is used only for ATTACH. You should choose the parameters for the macro instruction that provide the shortest search time. The search of a library actually involves the search of a directory, followed by copying the directory entry into virtual storage, followed by loading the load module into virtual storage. If you know the location of the load module, you should use parameters that eliminate as many of these searches as possible, as indicated in Figure 19, Figure 20, and Figure 21.

The EP, EPLOC, or DE parameter specifies the name of the entry point in the load module; you code one of the three every time you use a LINK, LOAD, XCTL, or ATTACH macro instruction. The optional DCB parameter indicates the address of the data control block for the library containing the load module. Omitting the DCB parameter or using the DCB parameter with an address of zero specifies the data control block for the task libraries, the job or step library, or the link library. If TASKLIB is specified and if the DCB parameter contains the address of the data control block for the link library, no other library is searched.

To avoid using "system copies" of modules resident in LPA and LINKLIB, you can specifically limit the search for the load module to the job pack area and the first library on the normal search sequence, by specifying the LSEARCH parameter on the LINK, LOAD, or XCTL macro instruction with the DCB for the library to be used.

The following paragraphs discuss the order of the search when the entry name used is a member name.

The EP and EPLOC parameters require the least effort on your part; you provide only the entry name, and the control program searches for a load module having that entry name. Figure 19 shows the order of the search when EP or EPLOC is coded, and the DCB parameter is omitted or DCB=0 is coded.

The job pack area is searched for an available copy.
The requesting task's task library and all the unique task libraries of its preceding tasks are searched. (Note: For the ATTACH macro, the attached task's library and all the unique task libraries of its preceding tasks are searched.)
The step library is searched; if there is no step library, the job library (if any) is searched.
The link pack area is searched.
The link library is searched.

Figure 19. Search for Module, EP or EPLOC Parameter With DCB=0 or DCB Parameter Omitted

When used without the DCB parameter, the EP and EPLOC parameters provide the easiest method of requesting a load module from the link, job, or step library. The task libraries are searched before the job or step library, beginning with the task library of the task that issued the request and continuing through the task libraries of all its antecedent tasks. The job or step library is then searched, followed by the link library.

A job, step, or link library or a data set in one of these libraries can be used to hold one version of a load module, while another can be used to hold another version with the same entry name. If one version is in the link library, you can ensure that the other will be found first by including it in the job or step library. However, if both versions are in the job or step library, you must define the data set that contains the version you want to use before the data set that contains the other version. For example, if the wanted version is in PDS1 and the unwanted version is in PDS2, a step library consisting of these data sets should be defined as follows:

```
//STEPLIB    DD  DSNAME=PDS1,...
//           DD  DSNAME=PDS2,...
```

If, however, the first version of a nonreusable module in the job or step library has been previously loaded and the version in the link library or the second version in the job library is desired, the DCB parameter must be coded in the macro instructions.

Use extreme caution when specifying module names in unique task libraries, because duplicate names may cause the wrong module to be brought into virtual storage when a task requests it. Once a module has been loaded from a task library, the module name is placed in the job pack queue. A copy of this module will then be available to all tasks in that job that request that module, regardless of the requester's task library.

If you know that the load module you are requesting is a member of one of the private libraries, you can still use the EP or EPLOC parameter, this time in conjunction with the DCB parameter. You specify the address of the data control block for the private library in the DCB parameter. The order of the search for EP or EPLOC with the DCB parameter is shown in Figure 20.

The job pack area is searched for an available copy.
The specified library is searched.
The link pack area is searched.
The link library is searched.

Figure 20. Search for Module, EP or EPLOC Parameters With DCB Parameter Specifying Private Library

Searching a job or step library slows the retrieval of load modules from the link library; to speed this retrieval, you should limit the size of the job and step libraries. You can best do this by eliminating the job library altogether and providing step libraries where required. You can limit each step library to the data sets required by a single step; some steps (such as compilation) do not require a step library and therefore do not require searching and retrieving modules from the link library. For maximum efficiency, you should define a job library only when a step library would be required for every step, and every step library would be the same.

The DE parameter requires more work than the EP and EPLOC parameters, but it can reduce the amount of time spent searching for a load module. Before you can use this parameter, you must use the BLDL macro instruction to obtain the directory entry for the module. The directory entry is part of the library that contains the module.

To save time, the BLDL macro instruction must obtain directory entries for more than one entry name. You specify the names of the load modules and the address of the data control block for the library when using the BLDL macro instruction; the control program places a copy of the directory entry for each entry name requested in a designated location in virtual storage. If you specify the link library and the job or step library, the directory information indicates from which library the directory entry was taken. The directory entry always indicates the relative track and block location of the load module in the library. If the load module is not located on the library you indicate, a return code is given. You can then issue another BLDL macro instruction specifying a different library.

To use the DE parameter, you provide the address of the directory entry and code or omit the DCB parameter to indicate the same library specified in the BLDL macro instruction. The task using the DE parameter should be the same as the one which issued the BLDL or one which has the same job, step, and task library structure as the task issuing the BLDL. The order of the search when the DE parameter is used is shown in Figure 21 for the link, job, step, and private libraries.

The preceding discussion of the search is based on the premise that the entry name you specified is the member name. The control program checks for an alias entry point name when the load module is found in a library. If the name is an alias, the control program obtains the corresponding member name from the library directory, and then searches to determine if a usable copy of the load module exists in the job pack area. If a usable copy does not exist in a pack area, a new copy is brought into the job pack area. Otherwise, the existing copy is used, conserving virtual storage and eliminating the loading time.

Directory Entry Indicates Link Library and DCB=0 or DCB Parameter Omitted.
  The job pack area is searched for an available copy.
  The link pack area is searched.
  The module is obtained from the link library.
Directory Entry Indicates Job, Step, or Task Library and DCB=0 or DCB Parameter Omitted.
  The job pack area is searched for an available copy.
  The module is obtained from the task library designated by the 'Z' byte of the DE operand.
DCB Parameter Indicates Private Library
  The job pack area is searched for an available copy.
  The module is obtained from the specified private library.

Figure 21. Search for Module Using DE Parameter

As the discussion of the search indicates, you should choose the parameters for the macro instruction that provide the shortest search time. The search of a library actually involves a search of the directory, followed by copying the directory entry into virtual storage, followed by loading the load module into virtual storage. If you know the location of the load module, you should use the parameters that eliminate as many of these unnecessary searches as possible, as indicated in Figure 19, Figure 20, and Figure 21. Examples of the use of these figures are shown in the following discussion of passing control.

Using an Existing Copy

The control program uses a copy of the load module already in the job pack area if the copy can be used. Whether the copy can be used or not depends on the reusability and current status of the load module; that is, the load module attributes, as designated using linkage editor control statements, and whether the load module has already been used or is in use. The status information is available to the control program only when you specify the load module entry name on an EXEC statement, or when you use ATTACH, LINK, or XCTL macro instructions to transfer control to the load module. The control program protects you from obtaining an unusable copy of a load module if you always "formally" request a copy using these macro

instructions (or the EXEC statement); if you pass control in any other manner (for instance, a branch or a CALL macro instruction), the control program, because it is not informed, cannot protect your copy.

All reenterable modules (modules designated as reenterable using the linkage editor) from any library are completely reusable; only one copy is ever placed in the link pack area or brought into your job pack area, and you get immediate control of the load module. If the module is serially reusable, only one copy is ever placed in the job pack area; this copy is always used for a LOAD macro instruction. If the copy is in use, however, and the request is made using a LINK, ATTACH, or XCTL macro instruction, the task requiring the load module is placed in a wait condition until the copy is available. A LINK macro instruction should not be issued for a serially reusable load module currently in use for the same task; the task will be abnormally terminated. (This could occur if an exit routine issued a LINK macro instruction for a load module in use by the main program.)

If the load module is not reusable, a LOAD macro instruction will always bring in a new copy of the load module; an existing copy is used only if a LINK, ATTACH, or XCTL macro instruction is issued and the copy has not been used previously. Remember, the control program can determine if a load module has been used or is in use only if all of your requests are made using LINK, ATTACH, or XCTL macro instructions.

Using the LOAD Macro Instruction

Use the LOAD macro instruction to ensure that a copy of the specified load module is in virtual storage. When a LOAD macro instruction is issued, the control program searches for the load module as discussed previously and, if required, brings a copy of the load module into virtual storage. When the control program returns control, register 0 contains the addressing mode and the virtual storage address of the entry point specified for the requested load module, and register 1 contains the length of the loaded module (in doublewords) and the authorization code in the high byte. Because the load module is retained even though it is not in use, the LOAD macro instruction is normally used only for a reenterable or serially reusable load module.

The control program brings the copy of the load module into subpool 251, with one exception. It places the module in subpool 252 when all of the following conditions exist:

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

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

The control program establishes a "responsibility" count for the copy, and adds one to the count each time the requirements of a LOAD macro instruction are satisfied by the same copy. As long as the responsibility count is not zero, the copy is retained in virtual storage.

The responsibility count for the copy is lowered by one when a DELETE macro instruction is issued during the task which was active when the LOAD macro instruction was issued. When a task is terminated, the count is lowered by the number of LOAD macro instructions issued for the copy when the task was active minus the number of deletions. When the use count for a copy in a job pack area reaches zero, the virtual storage area containing the copy is made available.

Passing Control with Return

The LINK macro instruction is used to pass control between load modules and to provide for return of control. You can also pass control using branch, branch and link, branch and save, or branch and save and set mode instructions or the CALL macro instruction; however, when you pass control in this manner you must protect against multiple uses of nonreusable or serially reusable modules. You must also be careful to enter the routine in the proper addressing mode. The following paragraphs discuss the requirements for passing control with return in each case.

The LINK Macro Instruction

When you use the LINK macro instruction, as far as the logic of your program is concerned, you are passing control to another load module. Remember, however, that you are requesting the control program to assist you in passing control. You are actually passing control to the control program, using an SVC instruction, and requesting the control program to find a copy of the load module and pass control to the entry point you designate. There is some similarity between passing control using a LINK macro instruction and passing control using a CALL macro instruction in a simple structure. These similarities are discussed first.

The convention regarding registers 2-12 still applies; the control program does not change the contents of these registers, and the called load module should restore them before control is returned. You must provide the address in register 13 of the save area for use by the called load module; the control program does not use this save area. You can pass address parameters in a parameter list to the load module using register 1; the LINK macro instruction provides the same facility for constructing this list as the CALL macro instruction. Register 0 is used by the control program and the contents may be modified. In certain cases, the contents of register 1 may be altered by the LINK macro instruction.

There is also some difference between passing control using a LINK macro instruction and passing control using a CALL macro instruction. When you pass control in a simple structure, register 15 contains the entry address and register 14 contains the return address. When the called load module gets control, that is still what registers 14 and 15 contain, but when you use the LINK macro instruction, it is the control program that establishes these addresses. When you code the LINK macro instruction, you provide the entry name and possibly some library information using the EP, EPLOC, or DE, and DCB parameters, but you have to get this entry name and library information to the control program. The expansion of the LINK macro instruction does this by creating a control program parameter list (the information required by the control program) and passing its address to the control program. After the control program finds the entry name, it places the address in register 15.

The return address in your control section is always the instruction following the LINK; that is not, however, the address that the called load module receives in register 14. The control program saves the address of the location in your program in its own save area, and places in register 14 the address of a routine within the control program that will receive control. Because control was passed using the control program, return must also be made using the control program. The control program also handles all switching of addressing mode when processing the LINK macro instruction.

The control program establishes a use count for a load module when control is passed using the LINK macro instruction. This is a separate use count from the count established for LOAD macro instructions, but it is used in the same manner. The count is increased by one when a LINK macro instruction is issued and decreased by one when return is made to the control program or when the called load module issues an XCTL macro instruction.

Figure 22 and Figure 23 show the coding of a LINK macro instruction used to pass control to an entry point in a load module. In Figure 22, the load module is from the link, job, or step library; in Figure 23, the module is from a private library. Except for the method used to pass control, this example is similar to Figures 10 and 11. A problem program parameter list containing the addresses INDCB, OUTDCB, and AREA is passed to the called load module; the return point is the instruction following the LINK macro instruction. A V-type address constant is not generated, because the load module containing the entry point NEXT is not to be edited into the calling load module. Note that the EP parameter is chosen, since the search begins with the job pack area and the appropriate library as shown in Figure 19.

```
            LINK    EP=NEXT,PARAM=(INDCB,OUTDCB,AREA),VL=1
RETURNPT    ...
AREA        DC      12F'0'
```

Figure 22. Use of the LINK Macro Instruction With the Job or Link Library

```
            OPEN    (PVTLIB)
            .
            .
            .
            LINK    EP=NEXT,DCB=PVTLIB,PARAM=(INDCB,OUTDCB,
                    AREA),VL=1
            .
            .
            .
PVTLIB      DCB     DDNAME=PVTLIBDD,DSORG=PO,MACRF=(R)
```

Figure 23. Use of the LINK Macro Instruction With a Private Library

Figure 24 and Figure 25 show the use of the BLDL and LINK macro instructions to pass control. Assuming that control is to be passed to an entry point in a load module from the link library, a BLDL macro instruction is issued to bring the directory entry for the member into virtual storage. (Remember, however, that time is saved only if more than one directory entry is requested in a BLDL macro instruction. Only one is requested here for simplicity.)

```
            BLDL    0,LISTADDR
            .
            .
            .
            DS      0H          List description field:
LISTADDR    DC      H'01'           Number of list entries
            DC      H'60'           Length of each entry
NAMEADDR    DC      CL8'NEXT'   Member name
            DS      26H         Area required for directory
                                information
```

Figure 24. Use of the BLDL Macro Instruction

The first parameter of the BLDL macro instruction is a zero, which indicates that the directory entry is on the link, job, step, or task library. The second parameter is the address in virtual storage of the list description field for the directory entry. The second two bytes at LISTADDR indicate the length of each entry. A character constant is established to contain the directory information to be placed there by the control program as a result of the BLDL macro instruction. The LINK macro instruction in Figure 25 can now be written. Note that the DE parameter refers to the name field, not the list description field, of the directory entry.

```
LINK     DE=NAMEADDR,DCB=0,PARAM=(INDCB,OUTDCB,AREA),VL=1
```

Figure 25. The LINK Macro Instruction With a DE Parameter

Using CALL or Branch and Link

You can save time by passing control to a load module without using the control program. Passing control without using the control program is performed as follows. Issue a LOAD macro instruction to obtain a copy of the load module, preceded by a BLDL macro instruction if you can shorten the search time by using it. The control program returns the address of the entry point and the addressing mode in register 0 and the length in doublewords in register 1. Load this address into register 15. The linkage requirements are the same when passing control between load modules as when passing control between control sections in the same load module: register 13 must contain a save area address, register 14 must contain the return address, and register 1 is used to pass parameters in a parameter list. A branch instruction, a branch and link instruction, a branch and save instruction, a branch and save and set mode instruction (BASSM), or a CALL macro instruction can be used to pass control, using register 15. Use BASSM only if there is to be an addressing mode switch. The return will be made directly to your program.

Notes:

1. *You must use a branch and save and set mode instruction if passing control to a module in a different addressing mode.*

2. *When control is passed to a load module without using the control program, you must check the load module attributes and current status of the copy yourself, and you must check the status in all succeeding uses of that load module during the job step, even when the control program is used to pass control.*

The reason you have to keep track of the usability of the load module has been discussed previously; you are not allowing the control program to determine whether you can use a particular copy of the load module. The following paragraphs discuss your responsibilities when using load modules with various attributes. You must always know what the reusability attribute of the load module is. If you do not know, you should not attempt to pass control yourself.

If the load module is reenterable, one copy of the load module is all that is ever required for a job step. You do not have to determine the status of the copy; it can always be used. You can pass control by using a CALL macro instruction, a branch, a branch and link instruction, a branch and save instruction, or a branch and save and set mode instruction (BASSM). Use BASSM only if there is to be an addressing mode switch.

If the load module is serially reusable, one use of the copy must be completed before the next use begins. If your job step consists of only one task, preventing simultaneous use of the same copy involves making sure that the logic of your program does not require a second use of the same load module before completion of the first use. An exit routine must not require the use of a serially reusable load module also required in the main program.

Preventing simultaneous use of the same copy when you have more than one task in the job step requires more effort on your part. You must still be sure that the logic of the program for each task does not require a second use of the same load module before completion of the first use. You must also be sure that no more than one task requires the use of the same copy of the load module at one time; the ENQ macro instruction can be used for this purpose. Properly used, the ENQ macro instruction prevents the use of a serially reusable resource, in this case a load module, by more than one task at a time. Refer to "Resource Control" for a complete discussion of the ENQ macro instruction. A conditional ENQ macro instruction can also be used to check for simultaneous use of a serially reusable resource within one task.

If the load module is nonreusable, each copy can only be used once; you must be sure that you use a new copy each time you require the load module. You can ensure that you always get a new copy by using a LINK macro instruction or by doing as follows:

1. Issue a LOAD macro instruction before you pass control.

2. Pass control using a branch, branch and link, branch and save, branch and save and set mode instruction, or a CALL macro instruction.

3. Issue a DELETE macro instruction as soon as you are through with the copy.

How Control is Returned

The return of control between load modules is the same as return of control between two control sections in the same load module. The program in the load module returning control is responsible for restoring registers 2-14, possibly loading a return code in register 15, passing control using the address in register 14 and possibly setting the correct addressing mode. The program in the load module to which control is returned can expect registers 2-13 to be unchanged, register 14 to contain the return address, and optionally, register 15 to contain a return code. Control can be returned using a branch instruction, a branch and set mode instruction or the RETURN macro instruction. If control was passed without using the control program, control returns directly to the calling program. However, if control was originally passed using the control program, control returns first to the control program, then to the calling program.

The action taken by the control program is as follows. The control program returns in the caller's addressing mode. When control was passed using a LINK or ATTACH macro instruction, the responsibility count was increased by one for the copy of the load module to which control was passed to ensure that the copy would be in virtual storage as long as it was required. The return of control indicates to the control program that this use of the copy is completed, and so the responsibility count is decreased by one. The virtual storage area containing the copy is made available when the responsibility count reaches zero.

Passing Control without Return

The XCTL macro instruction is used to pass control between load modules when no return of control is required. You can also pass control using a branch instruction; however, when you pass control in this manner, you must protect against multiple uses of nonreusable or serially reusable modules. The following paragraphs discuss the requirements for passing control without return in each case.

Passing Control Using a Branch Instruction

The same requirements and procedures for protecting against reuse of a nonreusable copy of a load module apply when passing control without return as were stated under "Passing Control With Return." The procedures for passing control are as follows.

A LOAD macro instruction should be issued to obtain a copy of the load module. The entry address and addressing mode returned in register 0 are loaded into register 15. The linkage requirements are the same when passing control between load modules as when passing control between control sections in the same load module; register 13 must be reloaded with the old save area address, then registers 14 and 2-12 restored from that old save area. Register 1 is used to pass parameters in a parameter list. If the addressing mode does not change, a branch instruction is issued to pass control to the address in register 15; if the addressing mode does change, a branch and save and set mode macro instruction is used.

Note: Mixing branch instructions and XCTL macro instructions is hazardous. The next topic explains why.

Using the XCTL Macro Instruction

The XCTL macro instruction, in addition to being used to pass control, is used to indicate to the control program that this use of the load module containing the XCTL macro instruction is completed. Because control is not to be returned, the address of the old save area must be reloaded into register 13. The return address must be loaded into register 14 from the old save area, as must the contents of registers 2-12. The XCTL macro instruction can be written to request the loading of registers 2-12, or you can do it yourself. If you restore all registers yourself, do not use the EP parameter. This creates an inline parameter list that can only be addressed using your base register, and your base register is no longer valid. If EP is used, you must have XCTL restore the base register for you.

When using the XCTL macro instruction, you pass parameters in a parameter list. In this case, however, the parameter list (or the parameter data) must be established in a portion of virtual storage outside the current load module containing the XCTL macro instruction. This is because the copy of the current load module may be deleted before the called load module can use the parameters, as explained in more detail below.

The XCTL macro instruction is similar to the LINK macro instruction in the method used to pass control: control is passed by way of the control program using a control program parameter list. The control program loads a copy of the load module, if necessary, loads the entry address in register 15, saves the address passed in register 14, and passes control to the address in register 15. The control program adds one to the responsibility count for the copy of the load module to which control is to be passed and subtracts one from the responsibility count for the current load module. The current load module in this case is the load module last given control using the control program in the performance of the active task. If you have been passing control between load modules without using the control program, chances are the responsibility count will be lowered for the wrong load module copy. And remember, when the

responsibility count of a copy reaches zero, that copy may be deleted, causing unpredictable results if you try to return control to it.

Figure 26 shows how this could happen. Control is given to load module A, which passes control to the load module B (step 1) using a LOAD macro instruction and a branch and link instruction. Register 14 at this time contains the address of the instruction following the branch and link. Load module B then is executed, independently of how control was passed, and issues an XCTL macro instruction when it is finished (step 2) to pass control to load module C. The control program knowing only of load module A, lowers the responsibility count of A by one, resulting in its deletion. Load module C is executed and returns to the address which used to follow the branch and link instruction. Step 3 of Figure 26 indicates the result.

Two methods are available for ensuring that the proper responsibility count is lowered. One way is to always use the control program to pass control with or without return. The other method is to use only LOAD and DELETE macro instructions to determine whether or not a copy of a load module should remain in virtual storage.

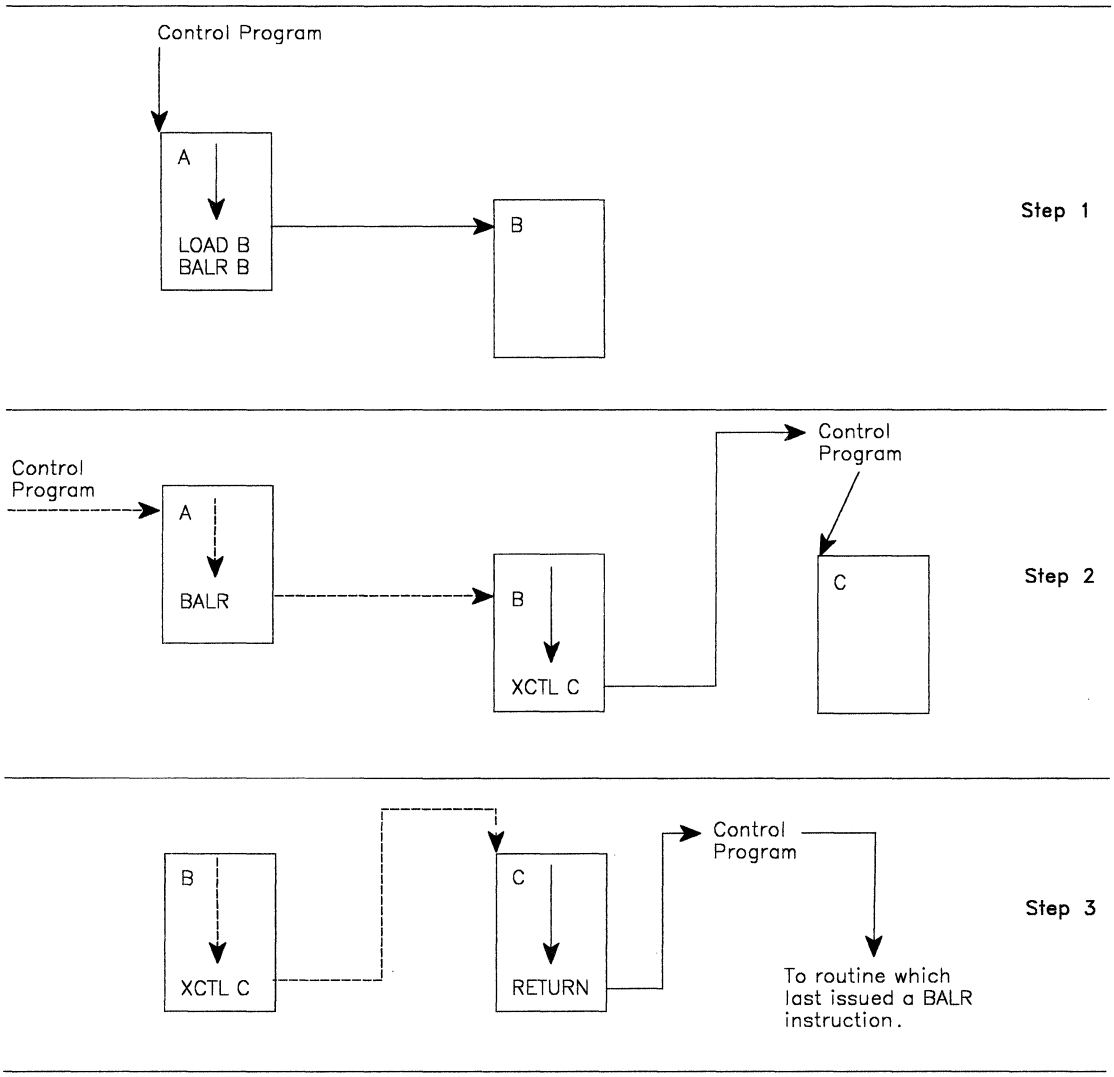

Figure 26. Misusing Control Program Facilities Causes Unpredictable Results

Additional Entry Points

Through the use of linkage editor facilities you can specify as many as 17 different names (a member name and 16 aliases) and associated entry points within a load module. It is only through the use of the member name or the aliases that a copy of the load module can be brought into virtual storage. Once a copy has been brought into virtual storage, however, additional entry points can be provided for the load module, subject to one restriction. The load module copy to which the entry point is to be added must be one of the following:

- A copy that satisfied the requirements of a LOAD macro instruction issued during the same task
- The copy of the load module most recently given control through the control program in performance of the same task

The entry point is added through the use of the IDENTIFY macro instruction, which can be issued only by a program running under a program request block (PRB). The IDENTIFY macro instruction cannot be issued by supervisor call routines or asynchronous exit routines established using other supervisor macro instructions.

When you use the IDENTIFY macro instruction, you specify the name to be used to identify the entry point, and the virtual storage address of the entry point in the copy of the load module. The address must be within a copy of an active load module that meets the requirements listed above; if it is not, the entry point will not be added, and you will be given a return code of 0C (hexadecimal). The name can be any valid symbol of up to eight characters, and does not have to correspond to a name or symbol within the load module. The name must not be the same as any other name used to identify any load module available to the control program; duplicate names cause errors. The control program checks the names of all active load modules in the link pack area, and the job pack area when you issue an IDENTIFY macro instruction, and provides a return code of 8 if a duplicate is found. You are responsible for not duplicating a member name or an alias in any of the libraries.

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

If an authorized caller creates an alias for a module in the pageable link pack area, IDENTIFY services places an entry for the alias on the active link pack area queue. If an unauthorized caller creates an alias for a module in the pageable link pack area, IDENTIFY services places an entry for the alias on the task's job pack queue.

Entry Point and Calling Sequence Identifiers as Debugging Aids

An entry point identifier is a character string of up to 70 characters that can be specified in a SAVE macro instruction. The character string is created as part of the SAVE macro instruction expansion.

A calling sequence identifier is a 16-bit binary number that can be specified in a CALL or a LINK macro instruction. When coded in a CALL or a LINK macro instruction, the calling sequence identifier is located in the two low-order bytes of the fullword at the return address. The high-order two bytes of the fullword form a NOP instruction.

Resource Control

When your program executes, other programs are executing concurrently in the MVS multiprogramming environment. Each group of programs, including yours, is a competitor for resources available at execution time. A competitor for resources is a *task*. As your program executes, it can identify another program as a task and request concurrent execution of that program.

A resource is anything that a program needs as it executes — such as processor time, a data set, another program, a table, or a hardware device, etc. If you subdivide an application into separate logical parts, and code it as several small programs instead of one large program, you can make the parts execute as separate tasks and with greater efficiency. However, you must ensure that each part executes in correct order relative to the others.

Use the macros in this chapter to introduce strategic delays into the code. These delays force programs to hold up execution of any sequence of code for which the necessary resources are not available, and commence execution as resources become available.

Task Synchronization

Some planning on your part is required to determine what portions of one task are dependent on the completion of portions of all other tasks. The POST macro instruction is used to signal completion of an event; the WAIT and EVENTS macro instructions are used to indicate that a task cannot proceed until one or more events have occurred. An event control block is used with the WAIT, EVENTS or POST macro instructions; it is a fullword on a fullword boundary, as shown in Figure 27.

An event control block is also used when the ECB parameter is coded in an ATTACH macro instruction. In this case the control program issues the POST macro instruction for the event (subtask termination). Either the 24-bit (bits 8 to 31) return code in register 15 (if the task completed normally) or the completion code specified in the ABEND macro instruction (if the task was abnormally terminated) is placed in the event control block as shown in Figure 27. The originating task can issue a WAIT or EVENTS WAIT=YES macro instruction specifying the event control block; the task will not regain control until after the event has taken place and the event control block is posted (except if an asynchronous event occurs, for example, timer expiration).

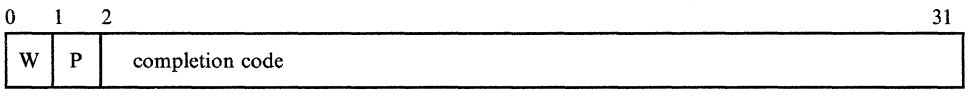

Figure 27. Event Control Block

When an event control block is originally created, bits 0 (wait bit) and 1 (post bit) must be set to zero. If an ECB is reused, bits 0 and 1 must be set to zero before a WAIT, EVENTS ECB= or POST macro instruction can be specified. If, however, the bits are set to zero before the ECB has been posted, any task waiting for that ECB to be posted will remain in the wait state. When a WAIT macro instruction is issued, bit 0 of the associated event control block is set to 1. When a POST macro instruction is issued, bit 1 of the associated event control block is set to 1 and bit 0 is set to 0. For an EVENTS type ECB, POST also puts the completed ECB address in the EVENTS table.

A WAIT macro instruction can specify more than one event by specifying more than one event control block. (Only one WAIT macro instruction can refer to a event control block at a time, however.) If more than one event control block is specified in a WAIT macro instruction, the WAIT macro instruction can also specify that all or only some of the events must occur before the task is taken out of the wait condition. When a sufficient number of events have taken place (event control blocks have been posted) to satisfy the number of events indicated in the WAIT macro instruction, the task is taken out of the wait condition.

An optional parameter, LONG=YES or NO, allows you to indicate whether the task is entering a long wait or a regular wait. A long wait should never be considered for I/O activity. However, you might want to use a long wait when waiting for an operator response to a WTOR macro instruction.

Using a Serially Reusable Resource

When one or more programs using a serially reusable resource modify the resource, they must not use the resource simultaneously with other programs. Consider a data area in virtual storage that is being used by programs associated with several tasks of a job step. Some of the programs are only reading records in the data area; because they are not updating the records, they can access the data area simultaneously. Other programs using the data area, however, are reading, updating, and replacing records in the data area. Each of these programs must serially acquire, update, and replace records by locking out other programs. In addition, none of the programs that are only reading the records want to use a record that another program is updating until after the record has been replaced.

If your program uses a serially reusable resource, you must prevent incorrect use of the resource. You must ensure that the logic of your program does not require the second use of the resource before completion of the first use. Be especially careful when using a serially reusable resource in an exit routine; because exit routines get control asynchronously with respect to your program logic, the exit routine could obtain a resource already in use by the main program. When more than one task is involved, using the ENQ macro instruction correctly can prevent simultaneous use of a serially reusable resource.

The ENQ macro instruction requests that the control program assign control of a resource to the active task. The control program determines the status of the resource, and does one of the following:

- If the resource is available, the control program grants the request by returning control to the active task.
- If the resource has been assigned to another task, the control program delays assignment of control by placing the active task in a wait condition until the resource becomes available.

- Passes back a return code indicating the status of the resource.
- Abends the caller on unconditional requests that would otherwise result in a non-zero return code.

When the status of the resource changes so that the waiting task can get control, the task is taken out of the wait condition and placed in the ready condition.

The DEQ macro instruction is used in conjunction with the ENQ macro instruction. If used properly, ENQ/DEQ can protect serially reusable resources. The rules for proper use of ENQ/DEQ are as follows:

- Everyone must use ENQ/DEQ.
- Everyone must use the same names and scope values for the same resources.
- Everyone must use consistent ENQ/DEQ protocol.

Naming the Resource

Represent the resource in the ENQ macro instruction by two names known as the qname and the rname, and by a scope indicator. The qname and rname need not have any relation to the actual name of the resource. The control program does not associate the name with the actual resource; it merely processes requests having the same qname, rname, and scope on a first-in, first-out basis. It is up to you to associate the names with the actual resource by ensuring that all users of the resource use qname, rname, and scope to represent the same resource. The control program treats requests having different qname, rname, and scope combinations as requests for different resources. Because the control program cannot determine the real name of the resource from the qname, rname, and scope, a task could use the resource by specifying a different qname, rname, and scope combination or by accessing the resource without using ENQ. In this case, the control program cannot provide any protection.

You will be abnormally terminated if you use SYSZ as the first four characters of a qname because the control program uses SYSZ for its qnames. Avoid using SYSA through SYSY because the control program sometimes uses these characters for its qnames as well. Either check with your system programmer to see which of the SYSA through SYSY combinations you can use or avoid using SYSx (where x is alphabetic) to begin qnames.

You can request a scope of STEP, SYSTEM, or SYSTEMS.

Use a scope of STEP if the resource is used only in your address space. The control program uses the address space identifier to make your resource unique in case someone else in another address space uses the same qname and rname and a scope of STEP.

Use a scope of SYSTEM if the resource is available to more than one address space in the system. All programs that serialize on the resource must use the same qname and rname and a scope of SYSTEM. For example, to prevent two jobs from using a named resource simultaneously, use SYSTEM.

Use a scope of SYSTEMS if the resource is available to more than one system. All programs that serialize on the resource must use the same qname and rname and a scope of SYSTEMS. For example, to prevent two processors from using a named resource simultaneously, use SYSTEMS. Note that the control program considers a resource with a SYSTEMS scope to be different from a resource represented by the same qname and rname but with a scope of STEP or SYSTEM.

Local and Global Resources

Global resource serialization, which handles ENQs and DEQs, recognizes two types of resources. These are local resources and global resources.

A local resource is a resource identified on the ENQ or DEQ macro instruction by a scope of STEP or SYSTEM. (Note that a resource with a scope of SYSTEM has its scope converted to SYSTEMS if the resource appears in the SYSTEM inclusion resource name list. See *Planning: Global Resource Serialization* for information about resource name lists.) A local resource is recognized and serialized only within the requesting operating system. The local resource queues are updated to reflect each request for a local resource. If a system is not operating under global resource serialization (that is, the system is not part of a global resource serialization complex), all resources requested are treated as local resources.

If a system is part of a global resource serialization complex, a global resource is identified on the ENQ or DEQ macro instruction by a scope of SYSTEMS. (Note that a resource with a scope of SYSTEMS has its scope changed to SYSTEM if the resource appears in the SYSTEMS exclusion resource name list.) A global resource is recognized and serialized by all systems in the global resource serialization complex.

Requesting Exclusive or Shared Control

To request exclusive control of the resource, code E in the ENQ macro instruction. If you are changing the resource, you must request exclusive control.

To request shared control of the resource, code S in the ENQ macro instruction. Request shared control only if you are not changing the resource.

Limiting Concurrent Requests for Resources

In order to prevent any one job, started task, or TSO user from generating too many concurrent requests for resources, global resource serialization counts and limits the number of ENQs in each address space. When a user issues an ENQ, global resource serialization increases the count of outstanding requests for that address space by one and decreases the count by one when the user issues a DEQ.

When the computed count reaches the threshold value or limit, global resource serialization processes subsequent requests as follows:

- Unconditional requests (ENQs that use the RET=NONE option) are abended with a system code of X'538'.
- Conditional requests (ENQs that specify the RET=HAVE or RET=USE option) are rejected and the user receives a return code of X'18'.

Processing the Request

The control program constructs a unique list for each qname, rname, and scope combination it receives in an ENQ macro instruction. When a task makes a request by issuing an ENQ macro instruction, the control program searches the existing lists for a matching qname, rname, and scope. If it finds a match, the control program adds the task's request to the end of the existing list; the list is not ordered by the priority of the tasks on it. If the control program does not find a match, it creates a new list, and adds the task's request as the first (and only) element. The task gets control of the resource based on the following:

- The position of the task's request on the list
- Whether or not the request was for exclusive or shared control

Figure 28 shows the status of a list built for a qname, rname, and scope combination. The S or E next to the entry indicates that the request was for either shared or exclusive control. The task represented by the first entry on the list always gets control of the resource, so the task represented by ENTRY1 (Figure 28, Step 1) is assigned the resource. The request that established ENTRY2 was for exclusive control, so the corresponding task is placed in the wait condition, along with the tasks represented by all the other entries in the list.

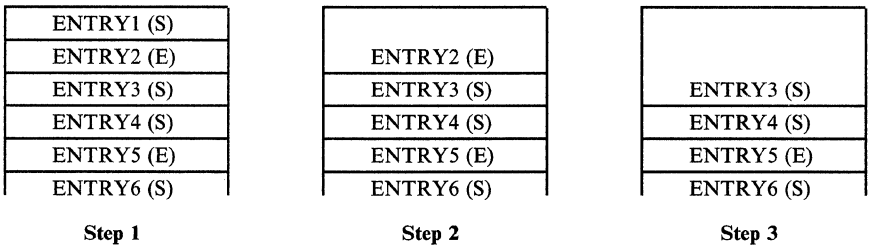

Figure 28. ENQ Macro Instruction Processing

Eventually, the task represented by ENTRY1 releases control of the resource, and the ENTRY1 is removed from the list. As shown in Figure 28, Step 2, ENTRY2 is now first on the list, and the corresponding task is assigned control of the resource. Because the request that established ENTRY2 was for exclusive control, the tasks represented by all the other entries in the list remain in the wait condition.

Figure 28, Step 3, shows the status of the list after the task represented by ENTRY2 releases the resource. Because ENTRY3 is now at the top of the list, the task represented by ENTRY3 gets control of the resource. ENTRY3 indicates that the resource can be shared, and, because ENTRY4 also indicates that the resource can be shared, ENTRY4 also gets control of the resource. In this case, the task represented by ENTRY5 does not get control of the resource until the tasks represented by ENTRY3 and ENTRY4 release control because ENTRY5 indicates exclusive use.

The control program uses the following general rules in manipulating the lists:

- The task represented by the first entry in the list always gets control of the resource.
- If the request is for exclusive control, the task is not given control of the resource until its request is the first entry in the list.
- If the request is for shared control, the task is given control either when its request is first in the list or when all the entries before it in the list also indicate a shared request.

- If the request is for several resources, the task is given control when all of the entries requesting exclusive control are first in their respective lists and all the entries requesting shared control are either first in their respective lists or are preceded only by entries requesting shared control.

Duplicate Requests for a Resource

A duplicate request occurs when a task issues an ENQ macro instruction to request a resource that the task already controls. For example, if a task that has control of a resource issues an unconditional ENQ macro to request the same resource, the task is abnormally terminated. If you make a duplicate request for a resource you might be abnormally terminated. With the second request, the control program recognizes the contradiction and returns control to the task with a non-zero return code or abnormally terminates the task. You should design your program to ensure that a second request for a resource made by the same task is never issued until control of the resource is released for the first use. Be especially careful when using an ENQ macro instruction in an exit routine. Two specific reasons why the use of ENQ in an exit routine must be carefully planned are:

- The exit may be entered more than once for the same TCB.
- An exit routine may request resources already obtained by some other process associated with the TCB.

More information on this topic follows under "Conditional and Unconditional Requests."

Releasing the Resource

Use the DEQ macro instruction to release a serially reusable resource that you obtained by using an ENQ macro instruction. If you try to release a resource for which you do not have control, you either get a non-zero return code or you are abnormally terminated. It is possible for many tasks to be placed in the wait condition while one task is assigned control of the resource. Having many tasks in the wait state might reduce the amount of work being done by the system, therefore, you should issue a DEQ macro instruction as soon as possible to release the resource, so that other tasks can use it. If a task terminates without releasing a resource, the control program releases the resource automatically.

Conditional and Unconditional Requests

Up to this point, only unconditional requests have been considered. You can, however, use the ENQ and DEQ macro instructions to make conditional requests by using the RET parameter. One reason for making a conditional request is to avoid the abnormal termination that occurs if you issue two ENQ macro instructions for the same resource within the same task or when a DEQ macro instruction is issued for a resource for which you do not have control.

The RET= parameter of ENQ and DEQ can provide the following options:

RET=CHNG	indicates the status of the resource specified is changed from shared to exclusive control.
RET=HAVE	indicates that control of the resource is requested conditionally; that is, control is requested only if a request has not been made previously for the same task.
RET=TEST	indicates the availability of the resource is to be tested, but control of the resource is not requested.
RET=USE	indicates control of the resource is to be assigned to the active task only if the resource is immediately available. If any of the resources are not available, the active task is not placed in a wait condition.

For the following descriptions, the term "active task" mean the task issuing the ENQ macro instruction. No reference is intended to different tasks which might be active in other processors of a multiprocessor.

RET=TEST is used by a task to test the status of the corresponding qname, rname, and scope combination, without changing the list in any way or waiting for the resource.

- A return code of 0 indicates that the active task does not now have control of the resource, but could have been given immediate control if it had been requested, because no other task has control of the resource.

- A return code of 4 indicates that another task has control of the resource, and the active task would have been placed in a wait condition if it had made an unconditional request.

- A return code of 8 indicates that the active task already has control of the resource.

- A return code of 14 indicates that the active task does not yet have control of the resource, but is in the list to be given control at a later time when other task(s) release the resource.

  Note: For return code 14 to occur, the restricted use of the ECB= parameter of the ENQ must have been used to make an entry on the list without placing the task in a wait condition.

RET=TEST is most useful for determining if the task already has control of the resource. It is less useful for determining the status of the list and taking action based on that status. In the interval between the time the control program checks the status and the time your program checks the return code and issues another ENQ macro instruction, another task could have been made active, and the status of the list could have changed.

RET=USE is used if you want your task to assigned control of the resource only if the resource is immediately available. If the resource is not immediately available, no entry will be made on the list and the task will not be made to wait. RET=USE is most useful when there is other processing that can be done without using the resource. For example, by issuing a preliminary ENQ with RET=USE in an interactive task, you can attempt to gain control of a needed resource without locking your terminal session. If the resource is not available, you can do other work rather than enter a long wait for the resource.

- A return code of 0 indicates that the active task did not have control of the resource prior to issuing the ENQ, but now has been given control and the corresponding entry has been put in the list.

- A return code of 4 indicates that the active task has not been given control of the resource, and an entry has not been made in the list, because another task already has control of the resource.

- A return code of 8 indicates that the active task already has control of the resource.

- A return code of 14 indicates that the active task does not yet have control of the resource, but is in the list to be given control at a later time when other task(s) release the resource.

- A return code of 18 indicates that the limit for the number of concurrent resource requests has been reached. The task does not have control of the resource unless some previous ENQ request caused the task to obtain control of the resource.

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

- A return code of 0 indicates that the active task now has exclusive control of the resource. Either exclusive control was already held, or shared control was converted to exclusive control as requested.
- A return code of 4 indicates that the requested change in attribute cannot be honored, because the active task is currently sharing the resource with another task.
- A return code of 8 indicates that the active task does not have an entry on the list for the specified resource. There is nothing to change.
- A return code of 14 indicates that the active task does have an entry on the list for the resource, but is not yet in control of the resource. No change is made.

RET=HAVE is used with both the ENQ and DEQ macro instructions to specify a conditional request for control of a resource (ENQ) when you do not know whether or not you have already requested control of that resource. RET=HAVE is used to release control (DEQ), with protection against abnormal termination of the active task, if an ENQ is duplicated or a DEQ is issued for a resource not held. If the resource is owned by another task, you will be put in a wait condition until the resource becomes available.

RET=HAVE with ENQ can make the active task wait until the resource becomes available.

- A return code of 0 indicates that the active task did not previously have an entry on the list or control of the resource, but has now been given control.
- A return code of 8 indicates that the active task already has control of the resource and already has an entry on the list. (Without RET=HAVE, this situation would cause abnormal termination. With RET=HAVE, it is effectively a no-operation.)
- A return code of 14 indicates that the active task has entry on the list for the resource, but is not yet in control of the resource. No change is made.

For DEQ:

- A return code of 0 indicates that the DEQ routine found an entry for the active task on the list for the specified resource, and has removed the entry. If the active task held control of the resource, this action relinquishes control. If the active task did not hold control of the resource (because the restricted ECB parameter had been used with ENQ, and control has not meanwhile become available), the DEQ routine simply removes the entry from the list without affecting control of the resource.

- A return code of 4 indicates the resource has been requested for the task, but the task has not been assigned control. The task is not removed from the wait condition. (This return code could result if DEQ is issued within an exit routine which was given control because of an interruption).

- A return code of 8 indicates that the active task did not have an entry on the list for the specified resource. There was no entry to dequeue.

If ENQ and DEQ are used in an asynchronous exit routine, code RET=HAVE to avoid possible abnormal termination.

Avoiding Interlock

An interlock condition happens when two tasks are waiting for each others' completion, but neither task can get the resource it needs to complete. Figure 29 shows an example of an interlock. Task A has exclusive access to resource M, and higher-priority task B has exclusive access to resource N. When task B requests exclusive access to resource M, B is placed in a wait state because task A has exclusive control of resource M.

The interlock becomes complete when task A requests exclusive control of resource N. The same interlock would have occurred if task B issued a single request for multiple resources M and N prior to task A's second request. The interlock would not have occurred if both tasks had issued single requests for multiple resources. Other tasks requiring either of the resources are also in a wait condition because of the interlock, although in this case they did not contribute to the conditions that caused the interlock.

```
       Task A                        Task B
ENQ (M,A,E,8,SYSTEM)
                              ENQ (N,B,E,8,SYSTEM)
                              ENQ (M,A,E,8,SYSTEM)
ENQ (N,B,E,8,SYSTEM)
```

Figure 29. Interlock Condition

The above example involving two tasks and two resources is a simple example of an interlock. The example could be expanded to cover many tasks and many resources. It is imperative that you avoid interlock. The following procedures indicate some ways of preventing interlocks.

- Do not request resources that you do not need immediately. If you can use the serially reusable resources one at a time, request them one at a time and release one before requesting the next.

- Share resources as much as possible. If the requests in the lists shown in Figure 29 had been shared, there would have been no interlock. This does not mean you should share a resource that you will modify. It does mean that you should analyze your requirements for the resources carefully, and not request exclusive control when shared control is enough.

- Use the ENQ macro instruction to request control of more than one resource at a time. The requesting program is placed in a wait condition until all of the requested resources are available. Those resources not being used by any other program immediately become exclusively available to the waiting program. For example, instead of coding the two ENQ macro instructions shown in Figure 30, you could code the one ENQ macro instruction shown in Figure 31. If all requests were made in this manner, the interlock shown in Figure 29 could not occur. All of the requests from one task would be processed before any of the requests from the second task. The DEQ macro instruction can release a resource as soon as it is no longer needed; resources requested in a multiple ENQ can be individually released through separate DEQ instructions.

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

Figure 30. Two Requests For Two Resources

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

Figure 31. One Request For Two Resources

- If the use of one resource always depends on the use of a second resource, then you can define the pair of resources as one resource in the ENQ and DEQ macro instructions. You can use this procedure for any number of resources that are always used in combination. However, the control program cannot protect these resources if they are also requested independently. Any requests must always be for the set of resources.

- If there are many users of a group of resources and some of the users require control of a second resource while retaining control of the first resource, it is still possible to avoid interlocks. In this case, each user should request control of the resources in the same order. For instance, if resources A, B, and C are required by many tasks, the requests should always be made in the order of A, B, and C. An interlock situation will not develop, since requests for resource A will always precede requests for resource B.

Program Interruption, Recovery/Termination, and Dumping Services

The supervisor offers many services 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, which causes the OPEN routine to request abnormal termination by issuing an ABEND macro instruction).

You can write exit routines to handle specific types of interruptions and abnormal conditions. The supervisor initiates the recovery/termination process for 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.

Interruption Services

Some conditions encountered in a program cause a program interruption. These conditions include incorrect parameters and parameter specifications, as well as exceptional results, and are known generally as program exceptions. You can disable the interruptions for certain exceptions (fixed point and decimal overflow, exponent underflow, and significance) by setting the corresponding bits in the program status word (PSW) to zero.

When a task becomes active for the first time, all program interruptions that can be disabled are disabled, and the task uses a standard control program exit routine, included when the system was generated. This exit routine gets control when certain program interruptions occur; it issues an ABEND macro instruction specifying task abnormal termination and requesting a dump.

Specifying User Exit Routines

By issuing the SPIE or ESPIE macro instruction, you can specify your own exit routine to be given control for one or more types of program exceptions. If you issue an ESPIE macro instruction, you can also pass the address of a parameter list to the exit routine. When one of the specified program exceptions occurs in a problem program being executed in the performance of a task, the exit routine receives control in the key of the TCB (TCBPKF) and in the addressing mode in effect when the SPIE or ESPIE was issued. (If a SPIE macro instruction was issued, this is 24-bit addressing mode.) For other program interruptions, part of the control program, the recovery termination manager (RTM), gets control. If the SPIE or ESPIE macro instruction specifies an exception for which the interruption has been disabled, the control program enables the interruption when the macro instruction is issued.

The environment established by an ESPIE macro instruction exists for the entire task, until the environment is changed by another SPIE/ESPIE macro instruction, or until the program creating the ESPIE returns via an SVC 3. Each succeeding SPIE or ESPIE macro instruction completely overrides specifications in the previous SPIE or ESPIE macro instruction. You can intermix SPIE and ESPIE macro instructions in one program. Only one SPIE or ESPIE environment is active at a time. If an exit routine issues a SPIE or ESPIE macro instruction, the new SPIE/ESPIE environment does not take effect until the exit routine completes.

The control program automatically deletes the SPIE/ESPIE exit routine when the RB that established the exit terminates. If a caller attempts to delete a specific SPIE/ESPIE environment established under a previous RB, the caller is abended with a system completion code of X'46D'. A caller can delete all previous SPIE and ESPIE environments (regardless of the RB under which they were established) by specifying a token of zero with the RESET option of the ESPIE macro instruction or an exit address of zero with the SPIE macro instruction.

Notes:

1. *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 representing the program that issued the macro instruction 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 life of the request block.*

2. *A SPIE exit routine established while executing in 24-bit addressing mode will not receive control if the program executing is in 31-bit addressing mode at the time of the interruption.*

Any problem program, executing in either 24-bit or 31-bit addressing mode in the performance of a task, can issue the ESPIE macro instruction. If your program is executing in 31-bit addressing mode, you cannot issue the SPIE macro instruction. The SPIE macro instruction is restricted in use to callers executing in 24-bit addressing mode in the performance of a task. The following topics describe how to use the SPIE and ESPIE macro instructions.

Using the SPIE Macro Instruction

The PICA and the program interruption element (PIE) contain the information that enables the control program to intercept user-specified program interruptions established using the SPIE macro instruction. The PIE and its associated PICA are called the "SPIE environment." You can modify the contents of the active PICA in order to change the active SPIE environment. The PICA and the PIE are described in the following topics.

Program Interruption Control Area

The expansion of each standard or list form of the SPIE macro instruction contains a control program parameter list called the program interruption control area (PICA). The PICA, as shown in Figure 32, contains the new program mask for the interruption types that can be disabled in the PSW, the address of the exit routine to be given control when one of the specified interruptions occurs, and a code for interruption types (exceptions) specified in the SPIE macro instruction.

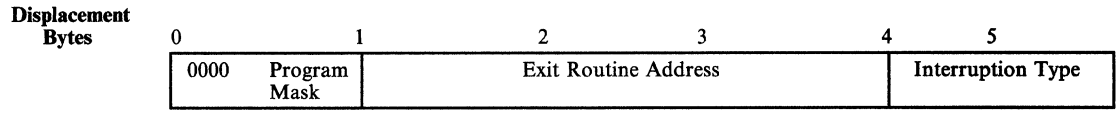

Figure 32. Program Interruption Control Area

The control program maintains a pointer (in the PIE) to the PICA referred to by the last SPIE macro instruction executed. This PICA might have been created by the last SPIE or might have been created previously and referred to by the last SPIE. Before returning control to the calling program or passing control to another program via an XCTL macro instruction, each program that issues a SPIE macro instruction must cause the control program to adjust the SPIE environment to the condition that existed previously or to eliminate the SPIE environment if one did not exist on entry to the program. When you issue the standard or execute form of the SPIE macro instruction, the control program returns the address of the previous PICA in register 1. If no SPIE/ESPIE environment existed when the program was entered, the control program returns zeroes in register 1.

You can cancel the effect of the last SPIE macro instruction by issuing a SPIE macro instruction with no parameters. This action does not reestablish the effect of the previous SPIE; it does create a new PICA that contains zeroes, thus indicating that you do not want an exit routine to process interruptions. You can reestablish any previous SPIE environment, regardless of the number or type of subsequent SPIE macro instructions issued, by using the execute form of the SPIE specifying the PICA address that the control program returned in register 1. The PICA whose address you specify must still be valid (not overlaid). If you specify zeroes as the PICA address, the SPIE environment is eliminated.

Figure 33 shows how to restore a previous PICA. The first SPIE macro instruction designates an exit routine called FIXUP that is to be given control if fixed-point overflow occurs. The address returned in register 1 is stored in the fullword called HOLD. At the end of the program, the execute form of the SPIE macro instruction is used to restore the previous PICA.

```
         .
         .
         .
         SPIE   FIXUP,(8)    Provide exit routine for fixed-point
                             overflow
         ST     1,HOLD       Save address returned in register 1
         .
         .
         .
         L      5,HOLD       Reload returned address
         SPIE   MF=(E,(5))   Use execute form and old PICA address
         .
         .
         .
HOLD     DC     F'0'
```

Figure 33. Using the SPIE Macro Instruction

Program Interruption Element

The first time you issue a SPIE macro instruction during the performance of a task, the control program creates a 32-byte program interruption element (PIE) in the virtual storage area assigned to your job step. Because the PIE is freed the first time you eliminate the SPIE environment (by specifying a PICA address of zero in the execute form of the SPIE macro instruction or by specifying a SPIE with no parameters), the control program also creates a PIE whenever you issue a SPIE macro instruction and no PIE exists. The format of the PIE is shown in Figure 34.

Hexadecimal Displacement (Bytes)	Decimal Displacement (Bytes)	1	2	3
0	0	Reserved	PICA Address	
4	4			(Interruption Codes)
C	12	Old Program Status Word in BC mode		
10	16	Register 14		
14	20	Register 15		
18	24	Register 0		
1C	28	Register 1		
20	32	Register 2		

Figure 34. Program Interruption Element

The PICA address in the PIE is the address of the program interruption control area used in the last execution of the SPIE macro instruction for the task. When control is passed to the routine indicated in the PICA, the BC mode old program status word contains the interruption code in bits 16-31 (the first byte is the exception extension code and the second is the exception code); you can test these bits to determine the cause of the program interruption. The control program stores the contents of registers 14,15,0,1, and 2 at the time of the interruption as indicated.

Using the ESPIE Macro Instruction

The ESPIE macro instruction extends the functions of the SPIE macro instruction to callers in 31-bit addressing mode. The options that you can specify using the ESPIE macro instruction are:

- SET to establish an ESPIE environment (that is, specify the interruptions for which the user-exit routine will receive control)
- RESET to delete the current ESPIE environment and restore the SPIE/ESPIE environment specified
- TEST to determine the active SPIE/ESPIE environment

If you specify ESPIE SET, you pass the following information to the service routine:

- A list of the program interruptions to be handled by the exit routine
- The location of the exit routine
- The location of a user-defined parameter list

The service routine returns a token representing the previously active SPIE or ESPIE environment, or zero if there was none.

If you code ESPIE RESET, you pass the token, which was returned when the ESPIE environment was established, back to the ESPIE service routine. The SPIE or ESPIE environment corresponding to the token is restored. If you pass a token of zero with **RESET**, all SPIE and ESPIE environments are deleted.

If you specify ESPIE TEST, you will be able to determine the active SPIE or ESPIE environment. An active SPIE environment is represented by a pointer to the PICA, which resides in user storage. (The PICA is described earlier in this section.) The active ESPIE environment is represented by protected control blocks belonging to the ESPIE service. To change an active ESPIE environment, you must issue the ESPIE macro with the SET or RESET option.

There are two control program areas associated with the ESPIE macro instruction. They are the extended program interruption element (EPIE) and the fake PICA. The EPIE and the fake PICA are described in the following topics.

The Extended Program Interruption Element (EPIE)

The control program creates an EPIE the first time you issue an ESPIE macro instruction during the performance of a task or whenever you issue an ESPIE macro instruction and no EPIE exists. The EPIE is freed when you eliminate the ESPIE environment.

The EPIE contains the information that the ESPIE service routine passes to the ESPIE exit routine when it receives control. When the exit routine receives control, register 1 contains the address of the EPIE. (See the topic "Register Contents Upon Entry to User's Exit Routine" for the contents of the other registers.) The format of the EPIE is shown in Figure 35.

Hexadecimal Displacement (Bytes)	Decimal Displacement (Bytes)	
0	0	'EPIE'
4	4	Address of user-supplied parameter list
8	8	Contents of the general purpose registers at the time of the interruption. The registers are stored in order-register 0 to register 15.
48	72	Old program status word in EC mode
50	80	Program interruption information consisting of the two-byte ILC followed by the two-byte interruption code
54	84	Address of a translation exception for a page fault (meaningful only if the interruption is a page fault)
58	88	Reserved

Figure 35. Extended Program Interruption Element

The Fake PICA

The fake PICA is used by MVS/XA to maintain compatibility between the SPIE and the ESPIE macro instructions. If you code a SPIE macro instruction to specify interruptions for which a SPIE exit routine is to receive control and if an ESPIE environment was previously active, the service routine returns the address of a fake PICA. The fake PICA resides in 24-bit addressable storage. The user should not modify its contents.

Register Contents Upon Entry to User's Exit Routine

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

Register 0:	Internal control program information.
Register 1:	Address of the PIE or EPIE for the task that caused the interruption.
Registers 2-12:	Same as when the program interruption occurred.
Register 13:	Address of the save area for the main program. The exit routine cannot use this area.
Register 14:	Return address (to the control program).
Register 15:	Address of the exit routine. The exit routine must be in virtual storage when it is required, and must return control to the control program using the address passed in register 14. For an ESPIE macro instruction, the control program restores all 16 registers from the EPIE. For a SPIE macro instruction, the control program restores registers 14,15,0,1, and 2 from the program interruption element after control is returned, but does not restore the contents of registers 3-13. If a program interruption occurs when the program interruption exit routine is in control, the control program exit routine gets control.

Functions Performed in User Exit Routines

Your exit recovery routine must determine the type of interruption that occurred before taking corrective action. Determining the type of interruption depends on whether the exit is associated with an ESPIE or a SPIE macro instruction.

- For an ESPIE, your exit recovery routine can check the two-byte interruption code (the first byte is the exception extension code and the second is the exception code) at offset X'52' in the EPIE.
- For a SPIE, your exit recovery routine can test bits 16 through 31 (the first byte is the exception extension code and the second is the exception code) of the old program status word (OPSW in BC mode) in the PIE.

Note: For both ESPIE and SPIE – If you are using vector instructions and an exception of 8, 12, 13, 14, or 15 occurs, your recovery routine can check the exception extension code (the first byte of the two-byte interruption code in the EPIE or PIE) to determine whether the exception was a vector or scalar type of exception. For more information about the exception extension code, see *IBM System/370 Vector Operations.*

Your recovery routine can alter the contents of the registers when control is returned to the interrupted program. The procedure for altering the registers also depends on whether the exit is associated with an ESPIE or a SPIE.

- For an ESPIE exit, the recovery routine can alter the contents of registers 0 through 15 in the save area in the EPIE because the control program reloads these registers from this area when it returns control to the interrupted program.
- For a SPIE exit, the recovery routine can alter registers 14 through 2 in the register save area in the PIE because the control program reloads these registers from this area when it returns control to the interrupted program. To change registers 3 through 13, the recovery routine must alter the contents of the registers.

The recovery routine can also alter the last four bytes of the OPSW in the PIE or EPIE. For an ESPIE, the recovery routine alters the CC and program mask starting at the third byte in the OPSW. By changing the OPSW, the routine can select any return point in the interrupted program. In addition, for ESPIE exits, the routine must set the AMODE bit of this four-byte address to indicate the addressing mode of the interrupted program.

Recovery/Termination Services

Part of the control program, the recovery termination manager (RTM), monitors the flow of control of software recovery processing and supplies the services of normal and abnormal task termination. RTM selects the appropriate recovery or termination process according to the status of the system.

RTM gets control in response to events such as the following:

- Unanticipated program checks (except those protected by SPIE recovery routines)
- Machine checks
- I/O error on page-in request
- ABEND macro instructions

RTM invokes any recovery routine that has been established to recover or clean up for the process in control. The recovery routine could be one of yours or it could be a system routine. If this recovery routine cannot recover from the incident (it requests termination or itself fails), RTM invokes the previously-established recovery routine. This passing of control from one recovery routine to another is called percolation. If none of the recovery routines can recover (request a retry), the control program terminates the process in control.

When a recovery routine gets control, it determines why it has been entered and decides either to percolate or to retry. To tell RTM what it wants done, the recovery routine issues the SETRP macro instruction, which manipulates fields in the system diagnostic work area (SDWA). When the recovery routine returns to RTM, RTM honors the request, if possible.

To allow communication between the main routine and the recovery routine, there is a parameter area. For a recovery routine established by an ESTAE macro instruction, you can supply a parameter area by coding the PARAM parameter on the macro instruction. When you establish a recovery routine, RTM saves a pointer to the parameter area and makes the pointer available to your recovery routine when it is entered. Usually, the main routine uses the parameter area to leave a footprint, that is, it sets indicators as part of normal processing; if an error occurs, these indicators 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.

If the recovery routine decides that a retry might be successful, it asks RTM to continue execution of the main routine at some appropriate point. Note that retry is not always allowed. If a recovery routine requests a retry when retry is not allowed, RTM ignores the request and continues with the termination process (percolates).

Any recovery routine that requests a retry must always include logic designed to avoid recursion, to prevent the creation of a tight loop between the recovery routine and the retry portion of the main 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 formed.

Using SETRP to Change the Completion and Reason Codes

You can specify both completion and reason code values on the ABEND macro instruction. RTM passes these values to recovery exit routines to identify abnormal terminations. You can change the values of the completion code and the reason code by using the SETRP macro instruction. The COMPCOD keyword allows you to specify a new completion code; the REASON keyword allows you to specify a new reason code.

The reason code has no meaning by itself, but must be used in conjunction with a completion code. In order to maintain meaningful completion and reason codes, RTM propagates changes to these values according to the following rules:

- If a user changes both the completion code and the reason code, RTM accepts both new values.
- If a user changes the reason code but not the completion code, RTM accepts the new reason code and uses the unchanged completion code.
- If a user changes the completion code but not the reason code, RTM accepts the new completion code and uses a zero for the reason code.
- If a user does not change either value, RTM uses the unchanged values.

Changing the Completion and Reason Codes Directly

Using the SETRP macro instruction is the preferred way for changing the completion and reason codes. If you change these values directly in a recovery exit routine you should emulate SETRP processing as follows:

- When you change the completion code, store the new completion code in SDWACMPC, a three-byte field in the system diagnostic word area (SDWA), and set the one-bit flag, SDWACCF, to indicate the change.
- When you change the reason code, store the new reason code in SDWACRC, a four-byte field in the SDWA, and set the one-bit flag, SDWAREAF, to indicate the change.

Before passing control to a recovery exit routine, RTM saves the current completion and reason codes. After the recovery routine returns control to RTM, RTM examines the contents of the SDWACCF and SDWAREAF flags to determine whether changes have been made to the completion and reason codes and then determines which values to pass to the next recovery exit routine. RTM makes this decision as shown in the following table:

SDWACCF Completion code flag	SDWAREAF Reason code flag	Values passed to the next recovery exit routine
ON	OFF	The abend completion code and a reason code of zero
OFF	ON	The unchanged completion code and the altered reason code
ON	ON	The altered completion code and the altered reason code

If both flags are off, RTM passes the values in the user's SDWA to the next recovery exit routine unless the completion code has been changed and the reason code has not been changed. In this case RTM passes the value of the completion code in the user's SDWA and a reason code of zero to the next recovery exit routine.

Handling ABENDs

The control program does a great deal of checking for abnormal conditions. It uses hardware to detect errors such as protection violations or addressing errors. The data management and supervisor routines provide some error checking facilities to ensure that, based on the information you have provided, only valid data is being processed, and that you have not made any conflicting requests. For abnormal conditions that can possibly be corrected, the control program returns to your program with a return code indicating the probable source of the error. For conditions that indicate that further processing would result in degradation of the system or destruction of data, the control program gives control to RTM.

There will, of course, be abnormal conditions unique to your program that the control program cannot detect. Figure 36 is an example of one of these. The routine shown in Figure 36 checks a control field in an input parameter list to determine which function the program is to perform. Only characters 1 through 4 are valid in the control field. The presence of any other character is invalid, but the routine must be prepared to detect and handle these characters. One way to handle an invalid character is to return to the calling program with an error return code. The calling program can then try to interpret the return code and recover from the error. If it cannot do so, the calling program can detach its incomplete subtasks, execute its usual termination procedures, and return control to its calling program, again with an error return code. This procedure might result in termination of all the tasks of a job step; if it does, you can use the COND parameters of the JOB and EXEC statements to indicate whether subsequent job steps should be executed.

Another way to handle this unexpected condition is to issue an ABEND macro instruction. RTM gets control.

The position within the job step hierarchy of the task for which the ABEND macro instruction is issued determines the exact function of the abnormal termination routine. If an ABEND macro instruction is issued when the job step task (the highest level or only task) is active, or if the STEP parameter is coded in an ABEND macro instruction issued during the performance of any task in the job step, all the tasks in the job step are terminated. For example, if the STEP parameter is coded in an ABEND macro under TSO, the TSO job will be terminated. An ABEND macro instruction (without a STEP parameter) that is issued in performance of any task in the job step task usually causes only that task and its subtasks to be abnormally terminated. However, if the abnormal termination cannot be fulfilled as requested, it might be necessary for RTM to abnormally terminate the job step task.

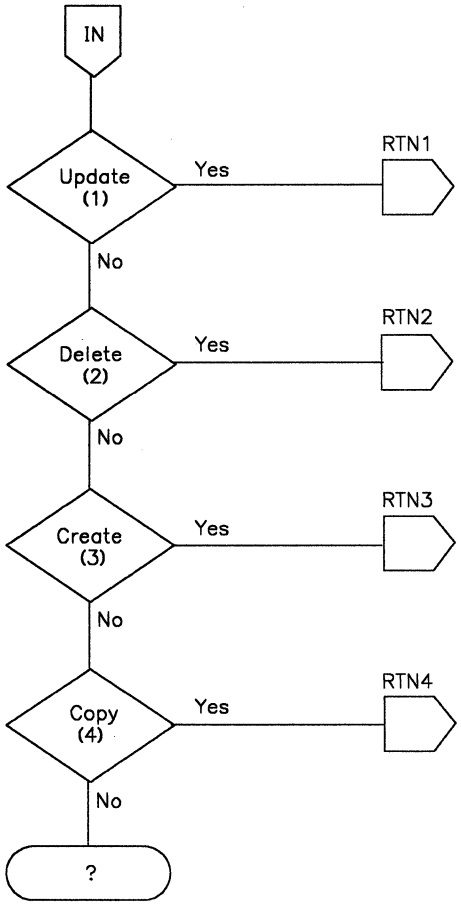

Figure 36. Detecting an Abnormal Condition

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

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

- Releases the resources owned by the terminating task and all of its subtasks starting with the lowest level task.
- Places the system or user completion code specified in the ABEND macro instruction in the task control block of the active task (the task for which the ABEND macro instruction was issued).
- Posts the ECB with the completion code specified in the ABEND macro instruction if the ECB parameter was coded in the ATTACH macro instruction issued to create the active task.

- Schedules the end-of-task exit routine to be given control when the originating task becomes active if the ETXR parameter was coded in the ATTACH macro instruction issued to create the active task.

- Calls a routine to FREEMAIN the terminating TCB.

If the job step is to be terminated, RTM:

- Releases the resources owned by each task, starting with the lowest level task, for all tasks in the job step. No end-of-task exit routine is given control.

- Writes the system or user completion code specified in the ABEND macro instruction on the system output device.

The remaining steps in the job are skipped unless you can establish your own recovery routine to perform similar functions and any other functions that your program requires. Use either the ESTAE macro instruction or the ATTACH macro instruction with the ESTAI option to set up an error routine that gets control whenever your program issues an ABEND macro instruction. Your error routine also gets control if the system issues an ABEND on your behalf. Your routine can determine its actions with regard to the abnormal condition. With this approach, you can put less error handling code in your mainline routines. For example, there is no need to check return codes after a subroutine if the subroutine issues an ABEND. The error handling functions can be part of the ESTAE or ESTAI routines that execute only when there is an error.

How to Use an ESTAE Recovery Routine

Within an ESTAE recovery routine, you can perform pre-termination functions and diagnose the error. You can also determine whether abnormal termination should continue for the task, or whether normal processing can continue at some point in the mainline routine.

When the abnormal termination is issued, the ESTAE recovery routine must be resident. It can either be part of the program issuing the ESTAE or be brought into virtual storage with the LOAD macro instruction.

A single program can create more than one recovery routine by issuing the ESTAE macro instruction with the CT parameter. (The program can also overlay or cancel recovery routines by issuing ESTAE macros with the OV parameter or with an address of zero, respectively.) All ESTAE requests issued by programs running under the same task are queued so that the routine established by the most recent ESTAE request is the first to get control. If this routine fails or requests that abnormal termination continue (percolation), RTM cancels the routine and the exit established by the previous ESTAE request gets control.

If you want to use the same recovery routine for several tasks at the same time, the routine must be reenterable. For convenience, you should make all your ESTAE exit routines reenterable.

You must cancel all the ESTAE routines you have created before returning control to your caller. If you try to cancel an ESTAE routine not associated with your request block, you get a return code that indicates your request is invalid.

Providing Information for Dump Analysis and Elimination

Dump analysis and elimination (DAE) uses information that callers provide in ESTAE 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. Each symptom string contains specific pieces of information called symptoms that DAE obtains from fields in the system diagnostic work area (SDWA), SDWA extensions, ABDUMP symptom area, and SDWA variable recording area (SDWAVRA).

When using DAE, you must select symptoms carefully. If the data you 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.

The following publications contain additional information pertaining to DAE:

- *SPL: System Modifications* provides information about how an installation can modify DAE to fit its needs.
- *Operations: System Commands* contains the syntax and use of the SET DAE command.
- *Debugging Handbook* contains sample symptom output and DAE control block information.
- *System Logic Library* provides a description of the logic and an explanation of symptom strings.
- *SPL: System Macros and Facilities Volume 1* describes the function of DAE for authorized users.

Interface to an ESTAE Recovery Routine

Before your first ESTAE recovery routine receives control, RTM performs I/O and asynchronous processing requests specified in the ESTAE macro instruction. RTM performs the requested I/O processing only for the first ESTAE routine. Subsequent routines receive an indication of the I/O processing previously done, but no additional processing is performed. However, RTM performs asynchronous processing for each routine.

The recovery routine is enabled and has the same protection key and PSW key mask (PKM) as the routine that established the recovery routine as long as the establishing routine was under a problem program protection key (keys 8-15). An ESTAE routine created by a program running under key 0-7 gets control in key 0.

Before each ESTAE recovery routine receives control, RTM tries to get storage for and to initialize a work area to contain information about the error. This work area is called the system diagnostic work area (SDWA). To access the SDWA, you must include the SDWA mapping macro - IHASDWA - as a DSECT in your ESTAE routine. Figure 37 shows key fields in the SDWA.

Field Name	Use
SDWAPARM	This four-byte field contains the pointer to the user parameter list that you supply for an ESTAE-type recovery routine.
SDWACMPC	This three-byte field contains the ABEND completion code that existed when RTM gave control to the recovery routine. The recovery routine can change the ABEND code by changing this field. The system code appears in the first twelve bits and the user code appears in the second twelve bits.
SDWAGRSV	This field shows the contents of general registers 0-15 as they were at the time of the error.
SDWACRC	This four-byte field contains the reason code that existed when RTM entered the recovery routine. The recovery routine can change the reason code by changing this field.
SDWAEC1	This field contains the PSW that existed at the time of the error.
SDWAEC2	The contents of this field vary according to the type of recovery routine: • For an ESTAE routine, the field contains the extended control PSW of the RB that created the recovery routine at the time the RB last incurred an interruption. • For an ESTAI routine, this field contains zeroes.
SDWASRSV	The contents of this field vary according to the type of recovery routine. • For an ESTAE routine, this field contains the general registers 0-15 of the RB that established the recovery routine as they were at the time the RB last incurred an interruption. • For an ESTAI routine, this field contains zeroes. If the recovery routine requests a retry, RTM uses the contents of this field to load the registers for the retry routine. To change the contents of the registers for the retry routine, you must make the changes to this field and request a register update on the SETRP macro instruction.
SDWASPID	This field contains the subpool ID of the SDWA.
SDWALNTH	This field contains the length, in bytes, of the SDWA.
SDWACOMU	The recovery routines use this 8-byte field to communicate with each other when percolation occurs. RTM copies this field from one SDWA to the next on all percolations. If the field contains zeroes, either there was no information passed or RTM was not able to pass it.
SDWAVRAL	This field contains the length of the variable recording area (VRA) for this SDWA.
SDWAHEX	This is a one bit field set by the recovery routine to indicate that EREP is to print the data in the VRA in hexadecimal form.
SDWAEBC	This is a one-bit field set by the recovery routine to indicate that EREP is to print the data in the VRA in EBCDIC form.
SDWAURAL	This is a one-byte field set by the recovery routine to indicate the length of the VRA used. The field initially contains zeroes. Whenever the recovery routine uses any part of the VRA, it must set this field.
SDWACCF	The recovery routine sets this one-bit field when it changes the completion code.
SDWAREAF	The recovery routine sets this one-bit field when it changes the reason code.
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.

Figure 37. Key Fields in the SDWA

The first field in the SDWA contains the address of the parameter list established by the ESTAE macro instruction. The register contents on entry to the ESTAE routine depends on whether or not RTM obtained an SDWA. If RTM obtained an SDWA, the registers are as follows:

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 I/O was active when the ABEND occurred. 16 — No I/O processing was performed.
Register 1	Address of the SDWA.
Registers 2-12	Unpredictable.
Register 13	Address of a 72-byte register save area.
Register 14	Return address.
Register 15	Entry point address.

When the ESTAE routine has completed its analysis of the error, it can use the SETRP macro instruction to inform RTM what it wants done. The SETRP macro instruction initializes the SDWA with the desired options. You can return from the ESTAE exit routine by using the SETRP REGS parameter or by using a BR 14 instruction.

If RTM could not obtain an SDWA, the register contents are as follows:

Register 0	12 (decimal). RTM could not obtain an SDWA.
Register 1	ABEND completion code.
Register 2	Address of the parameter list specified in the ESTAE macro instruction or 0.
Registers 3-13	Unpredictable.
Register 14	Return address.
Register 15	Entry point address.

If RTM could not provide an SDWA, it does not provide a register save area either. In this case, your ESTAE routine must save the address in register 14 and use it as the return address to RTM. You must place a return code in register 15 before returning to RTM. The return code indicates whether ABEND processing is to be continued for the task or whether a retry address can be given control. The return codes are:

Return Code	Meaning
0	Continue with termination. Any ESTAE routines that were established prior to this routine will get control.
4	Give control to the retry address. (You must place the retry address in register 0.)

How to Use an ESTAI Routine

You can provide an exit in your program to intercept abnormal termination of a subtask by using the ESTAI parameter on the ATTACH macro instruction you issue to create the subtask. Once you establish an ESTAI routine for one of your subtasks, it will be used for all of your subtasks. For example, suppose task A attaches task B and uses the ESTAI parameter in the ATTACH macro instruction. When task B attaches task C, the ESTAI routine created by task A is active for C as well as B.

Because more then one subtask can abnormally terminate at the same time, the ESTAI routine might be used by more than one subtask concurrently. Your ESTAI exit routines must therefore be reenterable.

Interface to an ESTAI Routine

ESTAI routines are entered after all ESTAE routines that exist for a given task have received control and have either failed or percolated. The interface to ESTAI routines is the same as for ESTAE exits, however, one additional option is available for ESTAI. When you return to RTM, you can specify return code 16 either on the SETRP macro instruction if an SDWA exists, or in register 15 if an SDWA is not available. The return code indicates to RTM that termination should continue and that no other ESTAI routines should receive control for that task.

ESTAE/ESTAI Retry Routines

If a given ESTAE or ESTAI routine requests percolation, RTM gives control to the next oldest ESTAE or ESTAI routine that exists for the task. However, if a given ESTAE or ESTAI exit routine requests retry, the control program takes a dump if requested and does not process any further ESTAE or ESTAI routines.

An ESTAE or ESTAI routine can request retry whenever the SDWACLUP bit in the SDWA is set to zero. To request retry, the exit routine must supply a retry address. The retry address is the point in the mainline 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 supplied. A retry routine requested by an ESTAE operates as an extension of the mainline code; it operates under the same RB and in the same addressing mode as the issuer of the ESTAE. All RBs prior to the retry RB are purged before giving control to the retry routine.

RTM purges the RB queue to cancel the effects of partially executed programs that are at a lower level in the program hierarchy than the program for which the retry occurs. Certain effects, however, cannot be canceled. Among these are:

- 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. You can use SVC RESTORE to have the control program restore all I/O requests on the list. The retry routine should free the storage occupied by the SDWA (if there was an SDWA) when that storage is no longer needed unless the exit routine specified FRESDWA=YES on the SETRP macro instruction. The subpool number and length to use on the FREEMAIN macro instruction are in the SDWA.

Interface to a Retry Routine

There are two different interfaces to a retry routine:

- If RTM was able to obtain an SDWA, you can set the register contents in the SDWA to whatever you wish and request that they be passed to the retry routine by coding RETREGS=YES in the SETRP macro instruction. This method is used most often in mainline processing.

- If RTM could not obtain an SDWA or if RETREGS=NO was specified on the SETRP macro instruction, only parameter registers are passed to the retry routine. This method is used most often if a special retry routine is to get control.

If RTM could not obtain an SDWA, register contents are as follows:

Register 0	12 (decimal)
Register 1	Address of the user parameter list established via the ESTAE macro instruction
Register 2	Address of the purge I/O restore list (PIRL) if I/O was quiesced and is restorable, otherwise 0
Registers 3-13	Unpredictable
Register 14	Address of an SVC 3 instruction
Register 15	Entry point address of the retry routine

If RTM obtained an SDWA and the retry routine specified RETREGS=NO or FRESDWA=NO:

Register 0	0
Register 1	Address of the SDWA
Registers 2-13	Unpredictable
Register 14	Address of an SVC 3 instruction
Register 15	Entry point address of the retry routine

If RTM obtained an SDWA and the retry routine specified RETREGS=NO and FRESDWA=YES:

Register 0	20 (decimal)
Register 1	Address of the user parameter list established via the ESTAE macro instruction
Register 2	Address of the PIRL if I/O was quiesced and is restorable, otherwise 0
Registers 3-13	Unpredictable
Register 14	Address of an SVC 3 instruction
Register 15	Entry point address of the retry routine

If the retry routine requested register update (RETREGS=YES), the registers, as they appear in the SDWA, are passed to the retry routine.

In all cases, the routine runs enabled and the protection key is the same key as the routine that established the retry routine.

Dumping Services

A problem program can request two types of storage dumps:

- An ABEND dump obtained through use of the DUMP parameter in the ABEND macro instruction or the DUMP=YES parameter on the SETRP macro instruction in a recovery exit.
- A snap dump obtained through use of the SNAP macro instruction.

ABEND Dumps

An ABEND macro instruction initiates error processing for a task. The DUMP option of ABEND requests a dump of storage and the DUMPOPT option may be used to specify the areas to be displayed. These dump options may be expanded by an ESTAE or ESTAI routine. The control program usually requests a dump for you when it issues an ABEND macro instruction. However, the control program can provide an ABEND dump only if you include a DD statement (SYSABEND, SYSMDUMP, or SYSUDUMP) in the job step. The DD statement determines the type of dump provided and the system dump options that are used. When the dump is taken, the dump options that you requested (specified in the ABEND macro instruction or by recovery routines) are added to the installation-selected options.

Note: The operator can use the CHNGDUMP command either to alter the dump options you or the installation specified, or to suppress all ABEND dumps.

If a dump is requested and the ESTAE/ESTAI routine also requests retry, the control program takes the dump before passing control to the retry address.

The data set containing the dump can reside on any device supported by the basic sequential access method (BSAM). The dump is placed in the data set described by the DD statement you provide. If you select a printer, the dump is printed immediately. However, if you select a direct access or tape device, you must schedule a separate job to obtain a listing of the dump, and to release the space on the device. If the dump data set was described by a SYSMDUMP DD statement, you can use the AMDPRDMP service aid to format and print the dump. (Do not select a printer for a SYSMDUMP DD statement.) For information about the AMDPRDMP service aid see *SPL: Service Aids.*

Obtaining a Symptom Dump

With all ABEND dumps, you will automatically receive a short symptom dump of approximately ten lines. This symptom dump provides a summary of error information, which will help you to identify duplicate problems.

You will receive this dump even without a DD statement unless your installation changes the default via the CHNGDUMP operator command or the dump parmlib member for SYSUDUMP.

SNAP Dumps

A task can request a SNAP dump at any time during its processing by issuing a SNAP macro instruction. For a SNAP dump, the DD statement can have any name except SYSABEND, SYSMDUMP, and SYSUDUMP.

Like the ABEND dump, the data set containing the dump can reside on any device that is supported by BSAM. The dump is placed in the data set described by the DD statement you provide. If you select a printer, the dump is printed immediately. However, if you select a direct access or tape device, you must schedule a separate job to obtain a listing of the dump, and to release the space on the device.

To obtain a dump using the SNAP macro instruction, you must provide a data control block and issue an OPEN macro instruction for the data set before issuing any SNAP macro instructions. If the standard dump format is requested, 120 characters per line are printed. The data control block must contain the following parameters: DSORG=PS, RECFM=VBA, MACRF=W, BLKSIZE=882 or 1632, and LRECL=125. (The data control block is described in *Data Management Services Guide* and *Data Management Macro Instructions.* If a high-density dump is to be printed on a 3800 Printing Subsystem, 204 characters per line are printed. To obtain a high-density dump, code CHARS=DUMP on the DD statement describing the dump data set. The BLKSIZE= must be either 1470 or 2724, and the LRECL= must be 209. CHARS=DUMP can also be coded on the DD statement describing a dump data set that will not be printed immediately. If CHARS=DUMP is specified and the output device is not a 3800, print lines are truncated and print data is lost. If your program is to be processed by the loader, you should also issue a CLOSE macro instruction for the SNAP data control block.

Finding Information in a SNAP Dump

You will obtain a dump index with each SNAP dump. The index will help you find information in the dump more quickly. Included in the information in the dump index is an alphabetical list of the active load modules in the dump along with the page number in the dump where each starts.

Obtaining a Summary Dump

You can request a summary dump for an abending task by coding the SUM option of the SNAP macro instruction. You can also obtain a summary dump by coding the DUMPOPT option of the ABEND or SETRP macro instruction and specifying a list form of SNAP that contains the SUM option.

If SUM is the only option that you specify, the dump will contain a dump header, control blocks, and the other areas listed below. The dump header for all ABEND dumps contains the following information:

- The dump title
- The ABEND code and program status word (PSW) at the time of the error
- If the PSW contains the address of an active load module:
  - The name and address from the PSW of the load module in error
  - The offset, into this load module, at which the error occurred

The control blocks and other areas consist of the following information:

- The control blocks dumped for the CB option
- The error control blocks (RTM2WAs and SCBs)
- The save areas
- The registers at the time of the error
- The contents of the load module (if the PSW contains the address of an active load module)
- The module pointed to by the last PRB (if it can be found)
- 1K of storage before and after the addresses pointed to by the PSW and the registers at the time of the error

  Note: This storage will only be dumped if the caller is authorized to obtain it. The storage is printed by ascending storage addresses, with duplicate addresses removed.

- The supervisor trace table consisting of all trace entries for the ASID that is being dumped.

  Note: The GTF trace records are not included in the dump.

If you specify other options in addition to SUM, the summary dump is dispersed throughout the dump. For example, if you specify the options:

```
CB,SUM,SPLS,LSQA,ERR,TRT
```

The dump will contain the following information:

- Dump title (SUM)
- ABEND code and PSW (SUM)
- Name and address from the PSW of the module in error (SUM)
- Offset into the module where the error occurred (SUM)
- Control blocks (SUM and CB)
- Error control blocks (ERR)
- Save areas (SUM)
- LSQA (LSQA)
- Registers at the time of the error (SUM)
- Contents of the active load module (SUM)
- 1K of storage before and after the addresses in the PSW and the registers at the time of the error (SUM)
- Subpool data (SPLS)
- If system trace is active, the system trace data and if GTF is active, the GTF trace data (TRT)

Virtual Storage Management

Use the virtual storage area assigned to your job step through implicit and explicit requests for virtual storage. The use of a LINK macro instruction is an example of an implicit request; the control program allocates storage before bringing the load module into your job pack area. The use of the GETMAIN macro instruction is an explicit request for a certain number of bytes of virtual storage to be allocated to the active task. In addition to your requests for virtual storage, requests are made by the control program and data management routines for areas to contain some of the control blocks required to manage your tasks.

Note: If your job step is to be executed as a nonpageable (V=R) task, the REGION parameter value specified on the job or execute statement determines the amount of virtual (real) storage reserved for the job step. If you run out of storage because of a system failure, such as in a GETMAIN request, increase the REGION parameter size.

The following paragraphs discuss some of the techniques that can be applied for efficient use of the virtual storage area reserved for your job step. These techniques apply as well to the data management portions of your programs. The specific data management storage allocation facilities are discussed in the *Data Management Services Guide* and *Data Management Macro Instructions* publications; the principles discussed here provide the background you need to use these facilities.

Explicit Requests for Virtual Storage

Virtual storage can be explicitly requested for the use of the active task by issuing a GETMAIN macro instruction. The request is satisfied by allocating a portion of the virtual storage area reserved for the job step. The virtual storage area is usually not set to zero when allocated. (The storage is zeroed for the initial allocation of a page).

You release virtual storage by issuing a FREEMAIN macro instruction. This does not release the area from control of the job step, but makes the area available to satisfy the requirements of additional requests for any task in the job step. The virtual storage assigned to a task is also given up to a different task in the same job step when the task terminates, except as indicated under "Subpool Handling." Releasing virtual storage for use by other job steps is discussed under "Relinquishing Virtual Storage."

Specifying the Size of the Area

Virtual storage areas are always allocated to the task in multiples of eight bytes and may begin on either a doubleword or page boundary. The request for virtual storage is given in terms of bytes; if the number specified is not a multiple of eight, it is rounded to the next higher multiple of eight. You can make repeated requests for a small number of bytes as you need the area or you can make one large request to completely satisfy the requirements of the task. There are two reasons for making one large request: it is the only way you can be sure of getting contiguous storage and avoid fragmenting your address space, and because you only make one request, the amount of control program overhead is less.

Types of Explicit Requests

There are several methods of explicitly requesting virtual storage using a GETMAIN macro instruction. Each method you select by using a parameter on GETMAIN, has certain advantages, depending on the requirements of your program. You can specify the actual location (above or below 16 Mb) of the virtual area allocated by using the LOC parameter of the GETMAIN macro instruction. (LOC is valid only with RU, RC, VRU, and VRC.) If you code LOC=ANY and the subpool indicated is supported above 16 Mb, GETMAIN attempts to allocate the virtual storage area above 16 Mb. If this is not possible or if the subpool is not supported above 16 Mb, GETMAIN tries to allocate the area below 16 Mb.

The last three methods do not produce reenterable coding unless coded in the list and execute forms. (See "Implicit Requests for Virtual Storage" on page 77 for additional information.) When you use the last three types, you can allocate storage below 16 Mb only.

The methods and the characters associated with them follow:

Register Type: There are several kinds of register requests. In each case the address of the area is returned in register 1. All of the register requests produce reenterable code because the parameters are passed to the control program in registers, not in a parameter list. The register requests are as follows:

R	specifies a request for a single area of virtual storage of a specified length, located below 16 Mb.
RU or RC	specifies a request for a single area of virtual storage of a specified length, located above or below 16 Mb according to the LOC parameter.
VRU or VRC	specifies a request for a single area of virtual storage with length between two values that you specify, located above or below 16 Mb according to the LOC parameter. GETMAIN attempts to allocate the maximum length you specify. If not enough storage is available to allocate the maximum length, GETMAIN allocates the largest area with a length between the two values that you specified. GETMAIN returns the length in register 0.

Element Type: EC or EU specifies a request for a single area of virtual storage, below 16 Mb, of a specified length. GETMAIN places the address of the allocated area in a fullword that you supply.

Variable Type: VC or VU specifies a request for a single area of virtual storage below 16 Mb with a length between two values you specify. GETMAIN attempts to allocate the maximum length you specify; if not enough storage is available to allocate the maximum length, the largest area with a length between the two values is allocated. GETMAIN places the address of the area and the length allocated in two consecutive fullwords that you supply.

List Type: LC or LU specifies a request for one or more areas of virtual storage, below 16 Mb, of specified lengths.

In addition to the above methods of requesting virtual storage, you can designate the request as conditional or unconditional. If the request is unconditional and sufficient virtual storage is not available to fill the request, the active task is abnormally terminated. If the request is conditional, however, and insufficient virtual storage is available, a return code of 4 is provided in register 15; a return code of 0 is provided if the request was satisfied.

An example of using the GETMAIN macro instruction is shown in Figure 38. The example assumes a program that operates most efficiently with a work area of 16,000 bytes, with a fair degree of efficiency with 8,000 bytes or more, inefficiently with less than 8,000 bytes. The program uses a reenterable load module having an entry name of REENTMOD, and will use it again later in the program; to save time, the load module was brought into the job pack area using a LOAD macro instruction so that it will be available when it is required.

```
            .
            .
            .
            GETMAIN EC,LV=16000,A=ANSWADD   Conditional request for 16,000 bytes
                                            in processor storage
            LTR     15,15                   Test return code
            BZ      PROCEED1                If 16,000 bytes allocated, proceed
            DELETE  EP=REENTMOD             If not, delete module and try to get
            GETMAIN VU,LA=SIZES,A=ANSWADD   smaller amount of virtual storage
            L       4,ANSWADD+4             Load and test allocated length
            CH      4,MIN                   If 8,000 or more, use procedure 1
            BNL     PROCEED1                If less than 8,000 use procedure 2
PROCEED2    ...
PROCEED1    ...
MIN         DC      H'8000'                 Min. size for procedure 1
SIZES       DC      F'4000'                 Min. size for procedure 2
            DC      F'16000'                Size of area for maximum efficiency
ANSWADD     DC      F'0'                    Address of allocated area
            DC      F'0'                    Size of allocated area
```

Figure 38. Using the GETMAIN Macro Instruction

A conditional request for a single element of storage with a length of 16,000 bytes is requested in Figure 38. The return code in register 15 is tested to determine if the storage is available; if the return code is 0 (the 16,000 bytes were allocated), control is passed to the processing routine. If sufficient storage is not available, an attempt to obtain more virtual storage is made by issuing a DELETE macro instruction to free the area occupied by the load module REENTMOD. A second GETMAIN macro instruction is issued, this time an unconditional request for an area between 4,000 and 16,000 bytes in length. If the minimum size is not available, the task is abnormally terminated. If at least 4,000 bytes are available, the task can continue. The size of the area actually allocated is determined, and one of the two procedures (efficient or inefficient) is given control.

Cell Pool Services

The cell pool macro instruction (CPOOL) 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 specified 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 provides the user with the following cell pool services:

- 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)

Subpool Handling

In an operating system, subpools of virtual storage are provided to assist in virtual storage management and for communications between tasks in the same job step. Because the use of subpools requires some knowledge of how the control program manages virtual storage, a discussion of virtual storage control is presented here.

Virtual Storage Control: When the job step is given a region of virtual storage, all of the storage area available for your use within that region is unassigned. Subpools are created only when a GETMAIN or CPOOL macro instruction is issued designating a subpool number (other than 0) not previously specified. If no subpool number is designated, the virtual storage is allocated from subpool 0, which is created for the job step by the control program when the job-step task is initiated.

The PSW key of the requestor is assigned to the subpool and does not change. A task, if it is authorized to do so, can change its PSW key. Such a change makes a previously acquired subpool unusable because the subpool's key no longer matches the task's key.

For purposes of control and virtual storage protection, the control program considers all virtual storage within the region in terms of 4096-byte blocks. These blocks are assigned to a subpool, and space within the blocks is allocated to a task by the control program when requests for virtual storage are made. When there is sufficient unallocated virtual storage within any block assigned to the designated subpool to fill a request, the virtual storage is allocated to the active task from that block. If there is insufficient unallocated virtual storage within any block assigned to the subpool, a new block (or blocks, depending on the size of the request) is assigned to the subpool, and the storage is allocated to the active task. The blocks assigned to a subpool are not necessarily contiguous unless they are assigned as a result of one request. Only blocks within the region reserved for the associated job step can be assigned to a subpool.

Figure 39 is a simplified view of a virtual-storage region containing four 4096-byte blocks of storage. All the requests are for virtual storage from subpool 0. The first request from some task in the job step is for 1008 bytes; the request is satisfied from the block shown as Block A in the figure. The second request, for 4000 bytes, is too large to be satisfied from the unused portion of Block A, so the control program assigns the next available block, Block B, to subpool 0, and allocates 4000 bytes from Block B to the active task. A third request is then received, this time for 2000 bytes. There is enough area in Block A (blocks are checked in the order first in, first out), so an additional 2000 bytes are allocated to the task from Block A. All blocks are searched for the closest fitting free area which will then be assigned. If the request had been for 96 bytes or less, it would have been allocated from Block B. Because all tasks may share subpool 0, Request 1 and Request 3 do not have to be made from the same task, even though the areas are contiguous and from the same 4096 byte block. Request 4, for 6000 bytes, requires that the control program allocate the area from 2 contiguous blocks which were previously unassigned, Block D and Block C. These blocks are assigned to subpool 0.

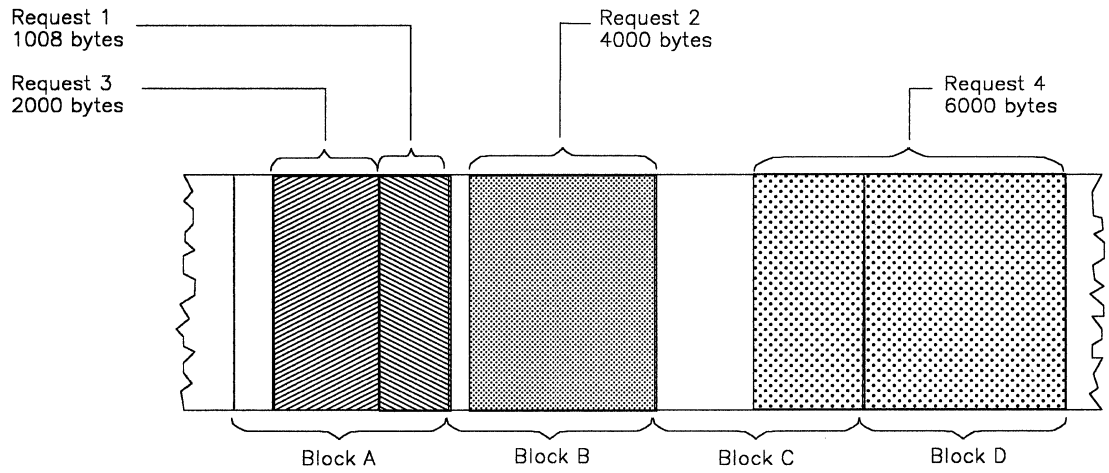

Figure 39. Virtual Storage Control

As indicated in the preceding example, it is possible for one 4096-byte block in subpool 0 to contain many small areas allocated to many different tasks in the job step, and it is possible that numerous blocks could be split up in this manner. Areas acquired by a task other than the job-step task are not released automatically on task termination. Even if FREEMAIN macro instructions were issued for each of the small areas before a task terminated, the probable result would be that many small unused areas would exist within each block while the control program would be continually assigning new blocks to satisfy new requests. To avoid this situation, you can define subpools for exclusive use by individual tasks.

Any subpool can be used exclusively by a single task or shared by several tasks. Each time that you create a task, you can specify which subpools are to be shared. Unlike other subpools, subpool 0 is shared by a task and its subtask, unless you specify otherwise. When subpool 0 is not shared, the control program creates a new subpool 0 for use by the subtask. As a result, both the task and its subtask can request storage from subpool 0 but both will not receive storage from the same 4096-byte block. When the subtask terminates, its virtual storage areas in subpool 0 are released; since no other tasks share this subpool, complete 4096-byte blocks are made available for reallocation.

Note: If the storage is shared, it is not released until the owning task terminates.

When there is a need to share subpool 0, you can define other subpools for exclusive use by individual tasks. When you first request storage from a subpool other than subpool 0, the control program assigns a new 4096-byte block to that subpool, and allocates storage from that block. The task that is then active is assigned ownership of the subpool and, therefore, of the block. When additional requests are made by the same task for the same subpool, the requests are satisfied by allocating areas from that block and as many additional blocks as are required. If another task is active when a request is made with the same subpool number, the control program assigns a new block to a new subpool, allocates storage from the new block, and assigns ownership of the new subpool to the second task.

A task can specify subpools numbered from 0 to 127. FREEMAIN macro instructions can be issued to release any complete subpool except subpool 0, thus releasing complete 4096-byte blocks. When a task terminates, its unshared subpools are released automatically.

Owning and Sharing: A subpool is initially owned by the task that was active when the subpool was created. The subpool can be shared with other tasks, and ownership of the subpool can be assigned to other tasks. Two macro instructions are used in the handling of subpools: the GETMAIN macro instruction and the ATTACH macro instruction. In the GETMAIN macro instruction, the SP parameter can be written to request storage from subpools 0 to 127; if this parameter is omitted, subpool 0 is assumed. The parameters that deal with subpools in the ATTACH macro instruction are:

- GSPV and GSPL, which give ownership of one or more subpools (other than subpool 0) to the task being created.
- SHSPV and SHSPL, which share ownership of one or more subpools (other than subpool 0) with the new subtask.
- SZERO, which determines whether subpool 0 is shared with the subtask.

All of these parameters are optional. If they are omitted, no subpools are given to the subtask, and only subpool 0 is shared.

Creating a Subpool: If the subpool specified does not exist for the active task, a new subpool is created whenever SHSPV or SHSPL is coded on an ATTACH macro instructions or a GETMAIN macro instruction is issued. A new subpool zero is created for the subtask if SZERO=NO is specified on ATTACH. If one of the ATTACH macro instruction parameters that specifies shared ownership of a subpool causes the subpool to be created, the subpool number is entered in the list of subpools owned by the task, but no blocks are assigned and no storage is actually allocated. If a GETMAIN macro instruction results in the creation of a subpool, the subpool number is assigned to one or more 4096-byte blocks, and the requested storage is allocated to the active task. In either case, ownership of the subpool belongs to the active task; if the subpool is created because of an ATTACH macro instruction, ownership is transferred or retained depending on the parameter used.

Transferring Ownership: An owning task gives ownership of a subpool to a direct subtask by using the GSPV or GSPL parameters in the ATTACH macro instruction issued when that subtask is created. Ownership of a subpool can be given to any subtask of any task, regardless of the control level of the two tasks involved and regardless of how ownership was obtained. A subpool cannot be shared with one or more subtasks and then transferred to another subtask, however; an attempt to do this results in abnormal termination of the active task. Ownership of a subpool can only be transferred if the active task has sole ownership; if the active task is sharing a subpool and an attempt is made to transfer it to a subtask, the subtask receives shared control and the originating task relinquishes the subpool. Once ownership is transferred

to a subtask or relinquished, any subsequent use of that subpool number by the originating task results in the creation of a new subpool. When a task that has ownership of one or more subpools terminates, all of the virtual storage areas in those subpools are released. Therefore, the task with ownership of a subpool should not terminate until all tasks or subtasks sharing the subpool have completed their use of the subpool.

If GSPV or GSPL specifies a subpool which does not exist for the active task, no action is taken.

Sharing a Subpool: Shared use of a subpool can be given to a direct subtask of any task with ownership or shared control of the subpool. Shared use is given by specifying the SHSPV or SHSPL parameters in the ATTACH macro instruction issued when the subtask is created. Any task with ownership or shared control of the subpool can add to or reduce the size of the subpool through the use of GETMAIN and FREEMAIN macro instructions. When a task that has shared control of the subpool terminates, the subpool is not affected.

Subpools in Task Communication: The advantage of subpools in virtual storage management is that, by assigning separate subpools to separate subtasks, the breakdown of virtual storage into small fragments is reduced. An additional benefit from the use of subpools can be realized in task communication. A subpool can be created for an originating task and all parameters to be passed to the subtask placed in the subpool. When the subtask is created, the ownership of the subpool can be passed to the subtask. After all parameters have been acquired by the subtask, a FREEMAIN macro instruction can be issued, under control of the subtask, to release the subpool virtual storage areas. In a similar manner, a second subpool can be created for the originating task, to be used as an answer area in the performance of the subtask. When the subtask is created, the subpool ownership would be shared with the subtask. Before the subtask is terminated, all parameters to be passed to the originating task are placed in the subpool area; when the subtask is terminated, the subpool is not released, and the originating task can acquire the parameters. After all parameters have been acquired for the originating task, a FREEMAIN macro instruction again makes the area available for reuse.

Implicit Requests for Virtual Storage

You make an implicit request for virtual storage every time you issue a LINK, LOAD, ATTACH, or XCTL macro instruction. In addition, you make an implicit request for virtual storage when you issue an OPEN macro instruction for a data set. This section discusses some of the techniques you can use to cut down on the amount of real storage required by a job step, and the assistance given you by the control program.

Reenterable Load Modules

A reenterable load module is designed so that during execution it does not modify itself. Only one copy of the load module is paged into real storage to satisfy the requirements of any number of tasks in a job step. This means that even though there are several tasks in the job step and each task concurrently uses the load module, the only real storage needed is an area large enough to hold one copy of the load module (plus a few bytes for control blocks). The same amount of real storage would be needed if the load module were serially reusable; however, the load module could not be used by more than one task at a time.

Reenterable Macro Instructions

All of the macro instructions described in this manual can be written in reenterable form. These macro instructions are classified as one of two types: macro instructions that pass parameters in registers 0 and 1, and macro instructions that pass parameters in a list. The macro instructions that pass parameters in registers present no problem in a reenterable program; when the macro instruction is coded, the required parameter values should be contained in registers. For example, the POST macro instruction requires that the ECB address be coded as follows:

```
POST ecb address
```

One method of coding this in a reenterable program would be to require this address to refer to a portion of storage that has been allocated to the active task through the use of a GETMAIN macro instruction. The address would change for each use of the load module. Therefore, you would load one of the general registers 2-12 with the address, and designate that register when you code the macro instruction. If register 4 contains the ECB address, the POST macro instruction is written as follows:

```
POST (4)
```

The macro instructions that pass parameters in a list require the use of special forms of the macro instruction when used in a reenterable program. The macro instructions that pass parameters in a list are identified within their descriptions in the macro instruction section of this manual. The expansion of the standard form of these macro instructions results in an in-line parameter list and executable instructions to branch around the list, to load the address of the list, and to pass control to the required control program routine. The expansions of the list and execute forms of the macro instruction simply divide the functions provided in the standard form expansion: the list form provides only the parameter list, and the execute form provides executable instructions to modify the list and pass control. You provide the instructions to load the address of the list into a register.

The list and execute forms of a macro instruction are used in conjunction to provide the same services available from the standard form of the macro instruction. The advantages of using list and execute forms are as follows:

- Any parameters that remain constant in every use of the macro instruction can be coded in the list form. These parameters can then be omitted in each of the execute forms of the macro instruction which use the list. This can save appreciable coding time when you use a macro instruction many times. (Any exceptions to this rule are listed in the description of the execute form of the applicable macro instruction.)

- The execute form of the macro instruction can modify any of the parameters previously designated. (Again, there are exceptions to this rule.)

- The list used by the execute form of the macro instruction can be located in a portion of virtual storage assigned to the task through the use of the GETMAIN macro instruction. This ensures that the program remains reenterable.

Figure 40 shows the use of the list and execute forms of a DEQ macro instruction in a reenterable program. The length of the list constructed by the list form of the macro instruction is obtained by subtracting two symbolic addresses; virtual storage is allocated and the list is moved into the allocated area. The execute form of the DEQ macro instruction does not modify any of the parameters in the list form. The list had to be moved to allocated storage because the control program can store a return code in the list when RET=HAVE is

coded. Note that the coding in a routine labeled MOVERTN is valid for lengths up to 256 bytes only. Some macro instructions do produce lists greater than 256 bytes when many parameters are coded (for example, OPEN and CLOSE with many data control blocks, or ENQ and DEQ with many resources), so in actual practice a length check should be made. The move long instruction (MVCL) should be considered for moving large data lists.

```
                .
                .
                .
                LA      3,MACNAME         Load address of list form
                LA      5,NSIADDR         Load address of end of list
                SR      5,3               Length to be moved in register 5
                BAL     14,MOVERTN        Go to routine to move list
                DEQ     ,MF=(E,(1))       Release allocated resource
                .
                .

* The MOVERTN allocates storage from subpool 0 and moves up to 256
* bytes into the allocated area.  Register 3 is from address,
* register 5 is length.  Area address returned in register 1.

MOVERTN         GETMAIN R,LV=(5)
                LR      4,1               Address of area in register 4
                BCTR    5,0               Subtract 1 from area length
                EX      5,MOVEINST        Move list to allocated area
                BR      14                Return
MOVEINST        MVC     0(0,4),0(3)
                .
                .
                .
MACNAME         DEQ     (NAME1,NAME2,8,SYSTEM),RET=HAVE,MF=L
NSIADDR         ...     ...
NAME1           DC      CL8'MAJOR'
NAME2           DC      CL8'MINOR'
```

Instruction in a Reenterable Program

Figure 40. Using the List and the Execute Forms of the DEQ Macro

Nonreenterable Load Modules

The use of reenterable load modules does not automatically conserve virtual storage; in many applications it will actually prove wasteful. If a load module is not used in many jobs and if it is not employed by more than one task in a job step, there is no reason to make the load module reenterable. The allocation of virtual storage for the purpose of moving coding from the load module to the allocated area is a waste of both time and virtual storage when only one task requires the use of the load module.

You should not make a load module reenterable or serially reusable if reusability is not really important to the logic of your program. Of course, if reusability is important, you can issue a LOAD macro instruction to load a reusable module, and later issue a DELETE macro instruction to release its area.

Notes:

1. *If your module is reenterable or serially reusable, the load module must be link edited, with the desired attribute, into a library.*

2. *A module that does not modify itself (a refreshable module) reduces paging activity because it does not need to be paged out.*

Freeing of Virtual Storage

The control program establishes two responsibility counts for every load module brought into virtual storage in response to your requests for that load module. The responsibility counts are lowered as follows:

- If the load module was requested in a LOAD macro instruction, that responsibility count is lowered when using a DELETE macro instruction.

- If the load module was requested in a LINK, ATTACH, or XCTL macro instruction, that responsibility count is lowered when using an XCTL macro instruction or by returning control to the control program.

- When a task is terminated, the responsibility counts are lowered by the number of requests for the load module made in LINK, LOAD, ATTACH, and XCTL macro instructions during the performance of that task, minus the number of deletions indicated above.

The virtual storage area occupied by a load module is released when the responsibility counts reach zero. When you plan your program, you can design the load modules to give you the best trade-off between execution time and efficient paging. If you use a load module many times in the course of a job step, issue a LOAD macro instruction to bring it into virtual storage; do not issue a DELETE macro instruction until the load module is no longer needed. Conversely, if a load module is used only once during the job step, or if its uses are widely separated, issue a LINK macro instruction to obtain the module and issue an XCTL from the module (or return control to the control program) after it has been executed.

There is a minor problem involved in the deletion of load modules containing data control blocks. An OPEN macro instruction must be issued before the data control block is used, and a CLOSE macro instruction issued when it is no longer needed. If you do not issue a CLOSE macro instruction for the data control block, the control program issues one for you when the task is terminated. However, if the load module containing the data control block has been removed from virtual storage, the attempt to issue the CLOSE macro instruction causes abnormal termination of the task. You must either issue the CLOSE macro instruction yourself before deleting the load module, or ensure that the data control block is still in virtual storage when the task is terminated (possibly by issuing a GETMAIN and creating the DCB in the area that had been allocated by the GETMAIN).

Data-in-Virtual

Data-in-virtual simplifies the writing of applications that use large amounts of data from permanent storage. Applications can create, read, and update data without the I/O buffer, blocksize, and record considerations that the traditional GET and PUT types of access methods require. Moreover, by using the services of data-in-virtual, certain applications that access large amounts of data can potentially improve their performance and their use of system resources.

Such applications have an accessing pattern that is non-sequential and unpredictable. This kind of pattern is a function of conditions and values that are revealed only in the course of the processing. In these applications, the sequential record subdivisions of conventional access methods are meaningless to the central processing algorithm. It is difficult to adapt this class of applications to conventional record management programming techniques, which require all permanent storage access to be fundamentally record-oriented. Through the DIV macro, data-in-virtual provides these applications with a way to manipulate the data without the constraints of record-oriented processing.

An application written for data-in-virtual views its permanent storage data as a seamless body of data without internal record boundaries. By using the data-in-virtual MAP service, the application can make any portion of the data set appear in virtual storage, in an area that is called a **virtual storage window**. Then, the data in the window can be referenced and updated with conventional processor instructions that access memory. When the application decides to copy the updates to the data set, it uses the data-in-virtual SAVE service.

The data-in-virtual services process the application data in 4096-byte units that are on page (4096-byte) boundaries. The processing is similar to that of paging data set support, but a special type of permanent data set is used to store the application data. This type of data set, which is called a **linear data set**, is supported by Access Method Services and is referred to in this book as a **data-in-virtual object,** a **data object,** or simply, an **object**. The data-in-virtual object is truly a continuous string of uninterrupted data.

When to Use Data-in-Virtual

When an application reads more input and writes more output data than necessary, the unnecessary reads and writes impact performance. You can expect improved performance from data-in-virtual because it reduces the amount of unnecessary I/O.

As an example of unnecessary I/O, consider the I/O performed by an interactive application that requires immediate access to several large data sets. The program knows that some of the data, although not all of it, will be accessed. However, the program does not know ahead of time which data will be accessed. A possible strategy for gaining immediate access to all the data is to read all the data ahead of time, reading each data set in its entirety insofar as this is possible. Once read into main storage, the data can be accessed quickly. However, if only a

small percentage of the data is likely to be accessed during any given period, the I/O performed on the unaccessed data is unnecessary.

Furthermore, if the application changes some data in main storage, it might not keep track of the changes. Therefore, to guarantee that all the changes are captured, the application must then write entire data sets back onto permanent storage even though only relatively few bytes are changed in the data sets.

Whenever such an application starts up, terminates, or accesses different data sets in an alternating manner, time is spent reading data that is not likely to be accessed. This time is essentially wasted, and the amount of it is proportional to the amount of unchanged data for which I/O is performed. Such applications are suitable candidates for a data-in-virtual implementation.

Factors Affecting Performance

When you write applications using the techniques of data-in-virtual, the I/O takes place only for the data referenced and the data changed and saved. If you implement an application using conventional access methods, and then implement it a second time using data-in-virtual techniques, you might notice an improvement in performance. The difference in performance depends on two factors: the **size** of the data set and its **access pattern** (or reference pattern). Size refers to the magnitude of the data sets that the application must process. The access pattern refers to whether the data being referenced is sequential or scattered throughout the data set.

To improve performance by using data-in-virtual, your data sets must be large **and** the data must be scattered throughout the data set.

Engineering and scientific applications often use data access patterns that are suitable for data-in-virtual. Among the applications that can be considered for a data-in-virtual implementation are:

- Applications that process large arrays
- VSAM relative record applications
- BDAM fixed length record applications

On the other hand, commercial applications often use data access patterns that are not suitable because they are predictable and sequential. If the access pattern of a proposed application is fundamentally sequential or if the data set is small, a conventional VSAM (or other sequential access method) implementation may perform better than a data-in-virtual implementation. However, this does not rule out commercial applications as data-in-virtual candidates. If the performance factors are favorable, any type of application, commercial or scientific, is suitable for a data-in-virtual implementation.

Creating a Linear Data Set

Before you can use the DIV macro to process a linear data set, the data set must exist on permanent storage.

To create the data set, you need to specify the DEFINE CLUSTER function of IDCAMS with the LINEAR parameter. When you code the SHAREOPTIONS parameter for DEFINE CLUSTER, the cross-system value must be 3; that is, you may code SHAREOPTIONS as (1,3), (2,3), (3,3), or (4,3). Normally, you should use SHAREOPTIONS (1,3). You can use the LOCVIEW parameter of the DIV macro in conjunction with the other SHAREOPTIONS.

LOCVIEW is described on page 87. For a complete explanation of SHAREOPTIONS, see the *VSAM Administration Guide.*

The following JCL invokes Access Method Services (IDCAMS) to create the linear data set named DIV.SAMPLE on the volume called DIVPAK. When IDCAMS creates the data set, it creates it as an empty data set. Note that no RECORDS parameter is used because linear data sets do not have records.

```
//JNAME    JOB 'ALLOCATE LINEAR',MSGLEVEL=(1,1),
//          CLASS=R,MSGCLASS=D,USER=JOHNDOE
//*
//*
//*         ALLOCATE A VSAM LINEAR DATASET
//*
//CLUSTPG  EXEC PGM=IDCAMS,REGION=4096K
//SYSPRINT DD SYSOUT=*
//DIVPAK   DD UNIT=3380,VOL=SER=DIVPAK,DISP=OLD
//SYSIN    DD *
           DEFINE CLUSTER (NAME(DIV.SAMPLE) -
                  VOLUMES(DIVPAK) -
                  TRACKS(1,1) -
                  SHAREOPTIONS(1,3) -
                  LINEAR)
/*
```

For further information on the creation of linear VSAM data sets and the alteration of entry-sequenced VSAM data sets, see *MVS/Extended Architecture Integrated Catalog Administration: Access Method Services Reference.*

Using the Services Of Data-in-Virtual

Each invocation of the DIV macro requests any one of eight distinct services provided by data-in-virtual:

- IDENTIFY
- ACCESS
- MAP
- SAVE
- RESET
- UNMAP
- UNACCESS
- UNIDENTIFY

Identify

To request processing of a data-in-virtual object, an application must invoke the IDENTIFY service. The application uses IDENTIFY to tell the system which data-in-virtual object it wants to process, via the DDNAME parameter. IDENTIFY generates a unique id, or token, that uniquely represents the application's individual request to use the given data set. It returns this id to the application. When the application requests other kinds of services with the DIV macro, it supplies this id as an input parameter.

Access

To gain the right to view or update the object, an application must use the ACCESS service. You normally invoke ACCESS after you invoke IDENTIFY and before you invoke MAP. ACCESS is similar to the OPEN macro of VSAM. It has a mode parameter of READ or UPDATE, and it gives your application the right to read or update the object.

If the object is a data set and if the SHAREOPTIONS parameter used to allocate the linear data set implies serialization, the system automatically serializes your access to the object. If access is not automatically serialized, you can serialize access to the object by using the ENQ, DEQ, and the RESERVE macros. If you do not serialize access to the object, you should consider using the LOCVIEW parameter to protect your window data against the unexpected changes that can occur when access to the object is not serialized. LOCVIEW is described on page 87.

Map

The data object is stored in units of 4096-byte blocks. An application uses the MAP service to specify the part of the object that is to be processed in virtual storage. It can specify the entire object (all of the blocks), or a part of the object (any continuous range of blocks).

After ACCESS, the application issues a GETMAIN macro to obtain a virtual storage area large enough to contain the string of blocks that is to be processed. The size of the object, which is returned (optionally) by ACCESS, can determine how much virtual storage needs to be requested when the application issues GETMAIN. After GETMAIN, the application invokes MAP. MAP establishes a direct correspondence between blocks on the object and pages in virtual storage. A continuous range of pages corresponds to a continuous range of blocks. This correspondence is called a **virtual storage window**, or a **window**.

After MAP, the application can look into the virtual storage area that is framed by the window. When it looks into this virtual storage area, it sees the same data that is on the object. When the application references this virtual storage area, it is referencing the data from the object, which is available in the window whenever a reference is made inside the window. To reference the area in the window, the application simply uses any conventional processor instructions that access memory.

Although the object data becomes available in the window when MAP is used, no actual movement of data from the object into the window occurs at that time. Actual movement of data from the object to the window can only occur when the application refers to data in the window. When a page in the window is referenced for the first time, a page fault occurs. When the page fault occurs, the system reads the permanent storage block into real storage.

When data is read into real storage, the data movement involves only the precise block that is referenced, which replaces the corresponding page in the window. Thus, only the blocks that are referenced by an application are ever read into real storage. The sole exception to this economical use of real storage occurs when the application specifies LOCVIEW=MAP with the ACCESS service for a data set object.

Notes:

1. *If the application specifies LOCVIEW=MAP with ACCESS, the entire window is immediately filled with object data when the application invokes MAP.*

2. *If, when an application invokes MAP, it would rather keep in the window the data that existed before the window was established (instead of having the object data appear in the window), it can specify RETAIN=YES. Specifying RETAIN=YES is useful when creating an object or overlaying the contents of an object.*

3. *An important concept for data-in-virtual is the concept of* **freshly obtained** *storage. When virtual storage has been obtained by a GETMAIN macro and not subsequently modified, the storage is considered to be in the freshly obtained state. When referring to this storage or any of its included pages, this book uses the terms, "freshly obtained storage" and "freshly obtained pages." If a program stores into a freshly obtained page, only that page loses its freshly obtained status, while other pages still retain it.*

Save and Reset

After the MAP service, the application can access the data inside the window directly through normal programming techniques. When the application changes some data in the window, however, the data on the object does not consequently change. If the application wants the data changes in the window to appear in the object, it must use the SAVE service. SAVE causes all changed blocks inside the window to be written out to the object. Unchanged blocks are not written. When SAVE completes, the object contains any changes that the application made inside the virtual storage window. If a SAVE is preceded by another SAVE, the second SAVE will only pick up the changes that occurred since the previous SAVE.

If the application changes some data in a virtual storage window but then decides not to keep those changes, it can use the RESET service to remove the changes from the window.

Unmap

When the application is finished processing the part of the object that is in the window, it eliminates the window by using UNMAP. To process a different part of the object, one not already mapped, the application can use the MAP service again. Because parts of the same object can be viewed simultaneously through several different windows, the application can set up separate windows on the same object. However, a specific page of virtual storage cannot be in more than one window at a time. The SAVE, RESET, MAP, and UNMAP services can be invoked repeatedly as required by the processing requirements of the application.

Unaccess

If the application has temporarily finished processing the object but still has other processing to perform, it uses UNACCESS to relinquish access to the object. Then, other programs can access the object. If the application needs to access the same object again, it can regain access to the object by using the ACCESS service again without having to use the IDENTIFY service again.

The application uses UNIDENTIFY to revoke its previous request, which was made by IDENTIFY, to process a data-in-virtual object.

The IDENTIFY Service

Your program uses IDENTIFY to select the data-in-virtual object that you want to process. IDENTIFY has three parameters: ID, DDNAME and TYPE.

Parameters of IDENTIFY - DDNAME: When you specify the IDENTIFY service, you specify a DDNAME parameter that identifies your data-in-virtual object. IDENTIFY causes the system to create a unique eight-byte id that it returns to you, stored in the address that you specify with the ID parameter. Your program supplies this id as a token input parameter on subsequent invocations of other data-in-virtual services.

Simultaneous requests for different processing activities against the same data-in-virtual object can originate from different tasks or from different routines within the same task. Each task or routine requesting processing activity against the data set must invoke the identify service. To correlate the various DIV macro invocations and processing activities, the eight-byte ids generated by IDENTIFY are sufficiently unique to reflect the individuality of the IDENTIFY request, yet they all reflect the same data-in-virtual object.

Parameters of IDENTIFY - TYPE: The TYPE parameter indicates the storage format of the data set name. You must specify TYPE=DA to indicate that the data set name is found in a data definition statement.

The ACCESS Service

Your program uses the ACCESS service to request permission to read or update the object. ACCESS uses three parameters: ID, MODE, and SIZE.

Parameters of ACCESS - ID: When you issue a DIV macro that requests the ACCESS service, you must also specify, on the ID parameter, the identifier that the IDENTIFY service returned. The ID parameter tells the system what object you want access to. When you request permission to access the object under a specified ID, the system checks the following conditions before it grants the access:

- The ID specified with your ACCESS request has already been established by a previous invocation of IDENTIFY.
- You have not already accessed the object under the same unique eight-byte ID. Before you can reaccess an already-accessed object under the same ID, you must first invoke UNACCESS for that ID.
- If your installation uses RACF, you must have the proper RACF authorization to access the object.
- If you are requesting read access, the object must not be empty. You use the MODE parameter to request read or update access.

Parameters of ACCESS - MODE: The MODE parameter specifies how your program will access the object. You can specify a mode parameter of READ or UPDATE. They are described as follows:

- READ lets you read the object, but prevents you from using SAVE to save changes to the object.
- UPDATE, like READ, lets you read the object, but it also allows you to update the object with SAVE.

Whether you specify READ or UPDATE, you can still make changes in the window without changing the object itself.

Parameters of ACCESS - SIZE: SIZE specifies the address of a four-byte field. When control is returned to your program after the ACCESS service executes, the four-byte field contains the current size of the object. The size is the number of blocks that must be mapped to ensure that the entire object is mapped. If SIZE is omitted or if SIZE=* is specified, the size is not returned.

Parameters of ACCESS - LOCVIEW: The LOCVIEW parameter allows you to specify whether the system is to create a local copy of the data-in-virtual object.

You can code the LOCVIEW parameter two ways:

- LOCVIEW=MAP
- LOCVIEW=NONE (the default if you do not specify LOCVIEW)

If another program maps the same block of a data-in-virtual object as your program has mapped, a change in the object due to a SAVE by the other program can sometimes appear in the virtual storage window of your program. The change can appear when you allocate the data set object with a SHAREOPTIONS(2,3), SHAREOPTIONS(3,3), or SHAREOPTIONS(4,3) parameter, and when the other program is updating the object while your program is accessing it.

If the change appears when your program is processing the data in the window, processing results might be erroneous, because the window data at the beginning of your processing is inconsistent with the window data at the end of your processing. For an explanation of SHAREOPTIONS, see *VSAM Administration Guide*. The relationship between SHAREOPTIONS and LOCVIEW is as follows:

- When you allocate the data set object by SHAREOPTIONS(2,3), SHAREOPTIONS(3,3), or SHAREOPTIONS(4,3), the system does not serialize the accesses that programs make to the object. Under these options, if the programs do not observe any serialization protocol, the data in your virtual storage window can change when other programs invoke SAVE. To ensure that your program has a consistent view of the object, and to protect your window from changes that other programs make on the object, use LOCVIEW=MAP. If you do not use LOCVIEW=MAP when you invoke ACCESS, the system provides a return code of 4 and a reason code of hexadecimal 37 as a reminder that no serialization is in effect even though the access was successful.
- When you allocate the object by SHAREOPTIONS(1,3), object changes made by the other program cannot appear in your window because the system performs automatic serialization of access. Thus, when any program has update access to the object, the system automatically prevents all other access. Use LOCVIEW=NONE when you allocate the data set by SHAREOPTIONS(1,3).

Note: The usual method of programming data-in-virtual is to use LOCVIEW=NONE and SHAREOPTIONS(1,3). LOCVIEW=MAP is provided for programs that must access a data object simultaneously. Those programs would not use SHAREOPTIONS(1,3).

LOCVIEW=MAP requires extra processing that degrades performance. Use LOCVIEW=NONE whenever possible, although you can use LOCVIEW=MAP for small data objects without significant performance loss. When you write a program that uses LOCVIEW=MAP, be careful about making changes in the object size. Consider the following:

- When a group of programs, all using LOCVIEW=MAP, have simultaneous access to the same object, no program should invoke any SAVE or MAP that extends or truncates the object, unless it informs the other programs by some coding protocol of a change in object size. When the other programs are informed, they can adjust their processing based on the new size.

- All the programs must create their maps before any program changes the object size. Subsequently, if any program wants to reset the map or create a new map, it must not do so without observing the protocol of a size check. If the size changed, the program should invoke UNACCESS, followed by ACCESS to get the new size. Then the program can reset the map or create the new map.

The MAP Service

The MAP service is used to make an association between part or all of a data object, the portion being specified by the OFFSET and SPAN parameters, and your program's virtual storage. This association, which is called a **virtual storage window,** takes the form of a one-to-one correspondence between specified blocks on the object and specified pages in the virtual storage. After the MAP is complete, your program can then proceed with the processing of the data in the window. The RETAIN parameter specifies whether data that is currently in the window is to be retained or to be replaced by the data from the associated object.

Note: Virtual storage pages that are page-fixed cannot be mapped into a virtual storage window. Once the window exists, you can page-fix data inside the window so long as it is not fixed when you issue SAVE or UNMAP.

The DIV macro has five parameters which are used when invoking MAP: ID, OFFSET, SPAN, AREA, and RETAIN.

Parameters of MAP - ID: The ID parameter specifies the storage location containing the unique eight-byte value that was returned by IDENTIFY. The map service uses this value to determine which object is being mapped under which request.

If you specify the same ID on multiple invocations of the MAP service, you can create simultaneous windows corresponding to different parts of the object. However, an object block that is mapped into one window cannot be mapped into any other window created under the same ID. If you use different IDs, however, an object block can be included simultaneously in several windows.

Parameters of MAP - OFFSET and SPAN: The OFFSET and SPAN parameters indicate a range of blocks on the object. Blocks in this range appear in the window. OFFSET indicates the first object block in the range, while SPAN indicates how many contiguous blocks make up the range. An offset of zero indicates the beginning of the object. For example, an offset of zero and a span of ten causes the block at the beginning of the object to appear in the window, together with the next nine object blocks. The window would then be ten pages long.

Specifying OFFSET=* or omitting OFFSET causes a default OFFSET of zero to be used. Specifying SPAN=0, SPAN=* or omitting SPAN results in a default SPAN of the number of blocks needed to MAP the entire object, starting from the block indicated by OFFSET. Specifying both OFFSET=* and SPAN=* or omitting both causes the entire object to appear in the window.

The OFFSET and SPAN parameters may be used to specify a range spanning any portion of the object, the entire object, or extending beyond the object. Specifying a range beyond the object enables a program to add data to the object, causing the object to grow. If data in a mapped range beyond the object is saved (using the SAVE service), the size of the object is updated to reflect the new size.

To use the OFFSET parameter, specify the storage location containing the block offset of the first block to be mapped. The offset of the first block in the data object is zero. To use the SPAN parameter, specify the storage location containing the number of blocks in the mapped range.

Parameters of MAP - AREA: When you specify MAP, you must also specify an AREA parameter. AREA indicates the beginning of a virtual storage space large enough to contain the entire window. Before invoking MAP, you must ensure that this virtual storage space is owned by your task. This can be ensured by specifying a storage area that is obtained by the GETMAIN macro. The storage must belong to a single, pageable private area subpool. It must begin on a 4096-byte boundary (that is, a page boundary) and have a length that is a multiple of 4096 bytes.

Note that any virtual storage space assigned to one window cannot be simultaneously assigned to another window. If your MAP request specifies, via the AREA parameter, a virtual storage location that is part of another window, the request is rejected.

A virtual storage area that is mapped into a window cannot be freed as long as the map exists. Attempts to do this will result in the virtual space being made unusable and an ABEND of the program. Subsequent attempts to reference the mapped virtual space will also result in an ABEND.

Parameters of MAP - RETAIN: The RETAIN parameter determines what data is to be viewed in the window. It can be either the contents of the virtual storage area (that corresponds to the window) the way it was before you invoked MAP, or it can be the contents of the object.

If you specify RETAIN=NO, or do not specify the RETAIN parameter at all (which defaults to RETAIN=NO), then the contents of the object replaces the contents of the virtual storage whenever a page in the window is referenced. Virtual storage that corresponds to a range beyond the end of the object appears as binary zeros when referenced. RETAIN=NO can be used when you want to view data already on the object, possibly with the intent of changing some data and saving it back to the object.

If you specify RETAIN=YES, then the window retains the contents of virtual storage. The data in the window are not replaced by data from the object as a result of the MAP operation. If you issue a subsequent SAVE, the data on the object is REPLACED by the data in the

window. If the window extends beyond the object and your program has not referenced the pages in the extending part of the window, then the extending pages are not saved. However, if the extending pages have been referenced, then they are saved on the object, causing the object to be extended so it can hold the additional data.

RETAIN=YES can also be used to reduce the size of (truncate) the object. If the part you want to truncate is mapped with RETAIN=YES and the window consists of freshly obtained storage, then at SAVE time, the data object size is reduced.

If you want to have zeroes written at the end of the object, then the corresponding virtual storage must be explicitly set to zero prior to the SAVE.

The SAVE Service

The SAVE service causes data to be written from virtual storage onto the data-in-virtual object. When you invoke SAVE, you specify a single and continuous range of blocks on the data-in-virtual object. Any virtual storage windows inside this range are eligible to participate in the SAVE.

For a SAVE request to be valid, the object must currently be accessed with MODE=UPDATE, under the same ID as the one specified on this SAVE request. Because an object can be mapped beyond its current end, the object might be extended when the SAVE is done if there are changed pages beyond the current end at the time of the ACCESS. On the other hand, the SAVE truncates the object if freshly obtained pages are mapped in a range that extends to or beyond the end of the object. In either case, the new object size is returned to your program if you specify the SIZE parameter.

Pages within the range cannot be page fixed when the SAVE is issued. Mapped pages in the range that are freshly obtained are written as binary zeroes if they occur before a changed page. The DIV macro has four parameters which are used when invoking SAVE: ID, OFFSET, SPAN and SIZE.

Notes:

1. *If data to be saved has not changed since the last SAVE, no I/O will be performed. The performance advantages of using data-in-virtual are primarily because of the automatic elimination of unnecessary read and write I/O operations.*

2. *The range specified with SAVE refers to the blocks of the object, which relate to pages inside a window.*

3. *The range specified with SAVE can extend beyond the end of the object.*

4. *Pages of the object not mapped to any window are not saved.*

5. *Pages in a window that lie outside the specified range are not saved.*

Parameters of SAVE - ID: The ID parameter tells the SAVE service which data object is being written to under which request. Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY. The object must have been previously accessed with MODE=UPDATE under the same ID as the one specified for SAVE.

Parameters of SAVE - OFFSET and SPAN: The OFFSET and SPAN parameters are used to select a continuous range of object blocks, within which the SAVE service can operate. OFFSET indicates the first block and SPAN indicates the number of blocks in the range. As in the MAP service, the offset and span parameters refer to object blocks; they do not refer to pages in the window.

Specifying OFFSET=* or omitting OFFSET causes the default offset (zero) to be used. The zero offset does not omit or skip over any of the object blocks, and it causes the range to start right at the beginning of the object. Specifying SPAN=0, SPAN=*, or omitting SPAN gives you the default span. The default span includes the first object block after the part skipped by the offset, and it includes the entire succession of object blocks up to and including the object block that corresponds to the last page of the last window.

When SAVE executes, it examines each virtual storage window established for the object. In each window, it detects every page that corresponds to an object block in the selected range. Then, if the page has changed since the last SAVE, the page is written on the object. (If the page has not changed since the last SAVE, it is already identical to the corresponding object block and there is no need for it to be saved.) Although SAVE discriminates between blocks on the basis of whether they have changed, it has the effect of saving all window pages that lie in the selected range. Specifying both OFFSET=* and SPAN=* or omitting both causes the system to save all changed pages in the window without exceptions.

To use the OFFSET parameter, specify the storage location containing the block offset of the first block to be saved. The Offset of the first block in the object is zero. To use the SPAN parameter, specify the storage location containing the number of blocks in the range to be saved.

Parameters of SAVE - SIZE: Invoking SAVE can change the size of the object. When you specify SIZE, the size of the object after SAVE completes is returned in the virtual storage location specified by the SIZE parameter. If you omit SIZE or specify SIZE=*, the size value is not returned.

Effect of RETAIN Mode on SAVE

While RETAIN cannot be specified on SAVE, the RETAIN mode of each included window affects how the window is saved.

When RETAIN=NO was specified for a previously mapped window, all pages in the virtual storage window appear to contain the contents of the object and are considered unchanged with respect to the object. When SAVE is issued, only pages in mapped windows that are changed and correspond to blocks within the range being saved are written to the object. Pages that might have been freshly obtained before the MAP, but have not been referenced since the MAP, still are not saved because they appear to contain the data that is on the corresponding blocks of the object. As with RETAIN=YES, the size of the object can still increase if the saved range extends beyond the current end of the object and it contains mapped windows with changed pages.

When RETAIN=YES was specified for a mapped virtual storage window, all pages in the window that are not freshly obtained are considered changed. SAVE writes all these pages, which are considered changed, onto the object (if they are within the range to be saved). If both the range to be saved and the previously mapped virtual storage window ranges extend

beyond the current end of the object, then the object is extended to hold the additional data. In addition, freshly obtained pages in the window will be written to the object as zeroes if:

- The freshly obtained pages correspond to blocks on the object within the current object size.
- The freshly obtained page currently being saved is followed, in the current window or in another window, by one or more changed pages.

On the other hand, freshly obtained pages in the window will cause the object to shrink in size (i.e. to be truncated) if:

- The freshly obtained page or pages are at the end of the window; that is, there are no changed pages beyond the last unchanged page.
- No other mapped windows beyond the current window contain changed pages.
- At least one of the freshly obtained pages in the window corresponds to the last block on the object; that is, the end of the object is mapped to a freshly obtained page. In this situation, the last block of the window must be equal to or greater than the last block of the object.

If all of the listed conditions for truncation are true, then the object is truncated to the last block that still contains data, either because (1) it corresponded to the last changed page in a mapped window or because (2) there was no mapped window page corresponding to that block. In the second case, the block is not affected by the SAVE; it contains the data that existed before the SAVE.

The RESET Service

At times during program processing, your program might have made changes to pages in the virtual storage window, and might no longer want to keep those changes. RESET, which is the opposite of SAVE, causes data from the object to appear in the virtual storage window. As with SAVE and MAP, the range to be reset refers to the object rather than the virtual storage. RESET only resets windows that are within the specified range, and it resets all the windows in the range.

Effect of RETAIN mode on RESET

Although the RETAIN parameter cannot be specified when you invoke RESET against a virtual storage window, the system remembers how RETAIN was specified on the MAP operation that created the window. Therefore, when you invoke RESET, the system resets the window one way for RETAIN=NO and another way for RETAIN=YES:

- If the window was created with RETAIN=NO, then after the RESET the data in the window is the same as the object data, as of the last SAVE. This applies to all the pages in the window.
- If the window was created with RETAIN=YES, then after the RESET the pages in the window acquire a freshly obtained status unless you have been doing SAVE operations on this window. Individual object blocks changed by those SAVE operations re-appear after

the RESET in their corresponding window pages, together with the other pages. However, the other pages appear freshly obtained.

Note: Regardless of the RETAIN mode of the window, any window page that corresponds to a block beyond the end of the object will appear as a freshly obtained page.

Parameters of RESET - ID: The ID parameter tells the RESET service what data object is being written to. Use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY and used with previous MAP requests. The object must be previously accessed (with either MODE=READ or MODE=UPDATE) under the same ID as the one being specified for RESET.

Parameters of RESET - OFFSET and SPAN: The OFFSET and SPAN keywords are used to indicate the RESET range, the part of the object that is to supply the data for the RESET. As with MAP and SAVE, OFFSET indicates the first object block in the range, while SPAN indicates how many contiguous blocks make up the range, starting from the block indicated by OFFSET. The first block of the object has an offset of zero.

To use OFFSET, specify the storage location containing the block offset of the first block to be reset. To use SPAN, specify the storage location containing the number of blocks in the range to be RESET. Specifying OFFSET=* or omitting OFFSET causes a default OFFSET of zero to be used. Specifying SPAN=* or omitting SPAN sets the default to the number of blocks needed to reset all the virtual storage windows that are mapped under the specified ID, however, starting only with the block number indicated by OFFSET. Specifying both OFFSET=* and SPAN=* or omitting both resets all windows that are currently mapped under the specified ID.

The UNMAP Service

Your program uses the UNMAP service to remove the association between a window in virtual storage and the object. Each UNMAP request must correspond to a previous MAP request. Note that, because UNMAP has no effect on the object, changes made in virtual storage but not yet saved are not saved on the object when UNMAP is issued. UNMAP has three parameters: ID, AREA, and RETAIN.

Parameters of UNMAP - ID: The ID parameter that you specify is the address of an eight-byte field in storage. That field contains the identifier associated with the object. The identifier is the same value that the IDENTIFY service returned, which is also the same value you specified when you issued the corresponding MAP request.

Parameters of UNMAP - AREA: The AREA parameter specifies the address of a four-byte field in storage that contains a pointer to the start of the virtual storage to be unmapped. This address must point to the beginning of a window. It is the same address that you provided when you issued the corresponding MAP request.

Parameters of UNMAP - RETAIN: RETAIN specifies the state that virtual storage is to be left in after it is unmapped; that is, after the correspondence between virtual storage and the object is removed.

Provided that you specify RETAIN=NO for the MAP service, specifying RETAIN=YES on the corresponding UNMAP transfers the object data that corresponds to unchanged pages into pages in the window. In this case, RETAIN=YES with UNMAP specifies that the virtual storage area corresponding to the unmapped window is to present to you the data that the

object contained, including any unsaved changes. After UNMAP, your program can still reference and change the data in this virtual storage, but can no longer save it on the object unless the virtual area is mapped again.

Specifying RETAIN=NO with UNMAP indicates that the data in the unmapped window is to be freshly obtained.

Notes:

1. *If RETAIN=YES is specified on MAP, RETAIN=YES on the corresponding UNMAP does not transfer the object value into the window. In this case, UNMAP leaves the window contents unchanged.*

2. *If UNMAP is issued with RETAIN=NO, and there are unsaved changes in the virtual storage window area, those changes are lost.*

3. *If unmap is issued with RETAIN=YES, and there are unsaved changes in thge window, they remain in the virtual storage.*

4. *The RETAIN parameter of MAP and UNMAP can be used to move data on the object without having to physically move it in virtual storage. For example, if you map block 2 to an area of virtual storage specifying RETAIN=NO, then block 2 of the object appears in the virtual storage window. If you then issue UNMAP with RETAIN=YES, block 2 is retained in virtual storage, even though the correspondence with the object has been removed.*

   Then, by mapping the virtual area to block 7 and specifying RETAIN=YES, you set up a correspondence between the same virtual area and block 7, with the original data from block 2 retained in the window. Finally, by issuing SAVE, you save the window (containing the block 2 data) on block 7 of the object. For more information on RETAIN, see the MAP and SAVE services. Note that the second map need not be for the same data object.

5. *Unmapping with RETAIN=YES has certain performance implications. It causes unreferenced pages, and maybe some unchanged ones, to be read from the object.*

The UNACCESS and UNIDENTIFY Services

You use UNACCESS to terminate your access to the object for the specified ID. UNACCESS automatically includes an implied UNMAP. If you issue an UNACCESS with outstanding virtual storage windows, any windows that exist for the specified ID are unmapped with RETAIN=NO. The ID parameter is the sole parameter of the UNACCESS service, and it designates the same ID that you specified in the corresponding ACCESS. As in the other services, use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY.

You use UNIDENTIFY to notify the system that your use of an object under the specified ID has ended. If the object is still accessed as an object under this ID, UNIDENTIFY automatically includes an implied UNACCESS. The UNACCESS, in turn, issues any necessary UNMAPs, using RETAIN=NO. The ID parameter is the only parameter for UNIDENTIFY, and it must designate the same ID as the one specified in the corresponding ACCESS. As in the other services, use ID to specify the storage location containing the unique eight-byte name that was returned by IDENTIFY.

Conditions for Invocation of Data-in-Virtual

The services of data-in-virtual execute synchronously, i.e. control is not returned from the DIV macro until the service is completed. Thus, before you can successfully invoke a service, the previous service must be complete. However, multiple services can be executing concurrently for different IDs. To invoke the DIV macro, you can be in the supervisor state, or you can be in the problem program state with a user key. You must supply a standard 72-byte save area pointed to by register 13. To issue the DIV macro, you must be:

1. enabled for I/O and external interrupts.
2. executing in 31 bit addressing mode.
3. executing in an environment that allows the use of SVCs. This means that you must be:
   - in task (TCB) mode.
   - not holding any locks.
   - in non-cross memory mode.

Sharing Services within a Task

The services of data-in-virtual are task-oriented. When a user issues IDENTIFY, an association is established between the ID assigned and the user's task. The type of association differs, however, depending on whether the task is authorized or not. The authorized task runs in supervisor state, has a system key (0-7), or has APF authorization. The non-authorized task runs in problem program state, with a user key, and with no APF authorization.

- For the non-authorized user, the use of data-in-virtual services for a specific ID is strictly local to its immediate task. That is, all services for a particular ID must be requested by the same task that requested IDENTIFY, and obtained the ID.

- For the authorized user, one task can issue IDENTIFY while authorized subtasks of that task, using the ID returned by IDENTIFY, can request all the other services, except ACCESS.

However, any task, authorized or not, may reference or change the data in a mapped virtual storage window, even if the window was mapped by another task, and even if the object was identified and accessed by another task. As long as the task can address the window, it is allowed to reference or change the included data.

Because data-in-virtual services affect virtual storage, the storage protect key of any task that requests a service (under a given ID) must be the same as the storage protect key of the task that issued the IDENTIFY (that obtained the ID). This is not required if the task has storage protect key zero.

Miscellaneous Restrictions

- When you attach a new task, you cannot pass ownership of a mapped virtual storage window to the new task. That is, the ATTACH keywords GSPV and GSPL cannot be used to pass the mapped virtual storage. Ownership of subpool zero cannot be passed using GSPV and GSPL keywords.

- Data-in-virtual services cannot be invoked in cross memory mode. There are no restrictions, however, against referencing and updating a mapped virtual storage window in cross memory mode.

DIV Macro Programming Examples

The programming examples in this section illustrate how to code and execute a program that processes a data-in-virtual object.

General Program Description

This is a description of the program shown in "Data-in-Virtual Sample Program Code" on page 97.

Step 1. The program issues a DIV IDENTIFY and DIV ACCESS for the data-in-virtual object. The ACCESS returns the current size of the object in units of 4K bytes.

Step 2. If the object contains any data (the size returned by ACCESS is non-zero), the program issues a DIV MAP to associate the object with storage the program acquires using GETMAIN. The size of the MAP (and the acquired storage area) is the same as the size of the object.

Step 3. The program now processes the input statements from SYSIN. The processing depends upon the function requests (S, D, or E). If the program encounters an end-of-file, the end-of-file is treated as if an "E" function was requested.

S function — Set a character in the object

Step 4. If the byte to change is past the end of the mapped area, the user asked to increase the size of the object. Therefore:

- Step a. If any changes have been made in the mapped virtual storage area but not saved to the object, the program issues a DIV SAVE. This save writes the changed 4K pages in the mapped storage to the object.
- Step b. The program issues a DIV UNMAP for the storage area acquired with GETMAIN, and then releases that area using FREEMAIN. The program skips this step if the current object size is 0.
- Step c. The program acquires storage using GETMAIN to hold the increased size of the object, and issues a DIV MAP for this storage.

Step 5. The program changes the associated byte in the mapped storage. Note that this does not change the object. The program actually writes the changes to the object when you issue a DIV SAVE.

D function — Display a character in the object

Step 6. If the requested location is within the MAP size, the program references the specified offset into the storage area. If the data is not already in storage, a page fault occurs. Data-in-virtual processing brings the required 4K block from the object into storage. Then the storage reference is re-executed. The contents of the virtual storage area (i.e. the contents of the object) are displayed.

E function — End the program

Step 7. If the program has made any changes in the mapped virtual storage area but has not saved them to the object, the program issues a DIV SAVE.

Step 8. The program issues a DIV UNIDENTIFY to terminate usage of the object. Note that data-in-virtual processing internally generates a DIV UNMAP and DIV UNACCESS.

Step 9. The program terminates.

Data-in-Virtual Sample Program Code

The first part of DIVSAMPL identifies the linear data set and accesses the object. If the object is not empty, the program obtains the virtual storage required to view (MAP) the entire object. Then it opens the input and message sequential data sets.

```
DIV      TITLE 'Data-in-Virtual Sample Program'
DIVSAMPL CSECT ,
DIVSAMPL AMODE 31                  Program runs in 31-bit mode
DIVSAMPL RMODE 24                  Program resides in 24-bit storage
         SAVE  (14,12),,'DIVSAMPL -- Sample Program'
         LR    R11,R15             Establish base register
         USING DIVSAMPL,R11        *
         LA    R2,VSVEAREA         Chain save areas together
         ST    R13,4(,R2)          *
         ST    R2,8(,R13)          *
         LR    R13,R2              *
* IDENTIFY and ACCESS the object pointed to by DD 'DIVDD'.
* Save the object's token in VTOKEN, and its size in VSIZEP.
         DIV   IDENTIFY,TYPE=DA,ID=VTOKEN,DDNAME=CDIVDD Specify DDNAME
         LA    R2,1                Error code
         LTR   R15,R15             IDENTIFY work ok ?
         BNZ   LERROR              * No -- quit
         DIV   ACCESS,ID=VTOKEN,MODE=UPDATE,SIZE=VSIZEP Open the object
         LA    R2,2                Error code
         LTR   R15,R15             ACCESS work ok ?
         BNZ   LERROR              * No -- quit
* If object not empty (VSIZEP > 0), get workarea to hold the object,
* and issue a MAP to it. The area must start on page boundary.
* Referencing byte "n" of this workarea gets byte "n" of the object.
         L     R2,VSIZEP           Current size (in 4K blocks)
         SLA   R2,12               Current size (in bytes)
         ST    R2,VSIZEB           VSIZEB = object size in bytes
         BZ    LEMPTY              If object not empty, get MAP area =
         GETMAIN RU,LV=(R2),LOC=(ANY,ANY),BNDRY=PAGE object size
         ST    R1,VAREAPTR         Save MAP area
         DIV   MAP,ID=VTOKEN,AREA=VAREAPTR,SPAN=VSIZEP
         LA    R2,3                Error code
         LTR   R15,R15             MAP work ok ?
         BNZ   LERROR              * No -- quit
LEMPTY   EQU   *                   Mapped, unless empty
* OPEN the SYSIN input data set, and SYSPRINT listing data set.
* Must be in 24-bit mode for this.  Then return to 31-bit mode.
         LA    R4,L31B01           Return to L31B01 in 31-bit mode
         LA    R1,L24B01           Go to L24B01 in 24-bit mode
         BSM   R4,R1               R4 = A(X'80000000'+L31B01)
L24B01   OPEN  (VSYSIN,(INPUT),VSYSPRT,(OUTPUT)) OPEN SYSIN/SYSPRINT
         BSM   0,R4                Return to 31-bit mode at next instr
L31B01   LA    R2,4                Error code from SYSIN OPEN
         LTR   R15,R15             OPEN ok ?
         BNZ   LERROR              * No -- quit
```

Data-in-Virtual Sample Program Code (continued)

The program reads statements from SYSIN until it reaches end-of-file, or encounters a statement with an "E" in column 1. The program validates the location in the object to set or display, and branches to the appropriate routine to process the request.

```
*
* Loop reading from SYSIN. Process the statements.
* Treat EOF as if the user specified "E" as the function to perform.
*
LREAD    EQU   *                    Read first/next card
         MVI   VCARDF,C'E'          EOF will appear as "E" function
         LA    R4,L31B02            Return to L31B02 in 31-bit mode
         LA    R1,L24B02            Go to L24B02 in 24-bit mode
         BSM   R4,R1                R4 = A(X'80000000'+L31B02)
L24B02   GET   VSYSIN,VCARD         Get the next input request.
LEOF     EQU   *                    End-of-file branches here
         BSM   0,R4                 Return to 31-bit mode at next instr
L31B02   EQU   *                    Get here in 31-bit mode
*
* Process request:
* E                      - End processing
* S aaaaaaaa v           - Set location X'aaaaaaaa' to v
* D aaaaaaaa             - Display location X'aaaaaaaa'
*
         CLI   VCARDF,C'E'          EOF function or EOF on data set ?
         BE    LCLOSE               * Yes -- go cleanup and terminate
         TRT   VCARDA,CTABTRT       Ensure A-F, 0-9
         BNZ   LINVADDV             * If not, is error
         MVC   VTEMP8,VCARDA        Save address
         TR    VTEMP8,CTABTR        Convert to X'0A'-X'0F', X'00'-X'09'
         PACK  VCHGADDR(5),VTEMP8(9) Make address
         L     R1,VCHGADDR          Address
         LA    R1,0(,R1)            Clear hi-bit
         ST    R1,VCHGADDR          Save address to change/display
         CLI   VCARDF,C'D'          Display requested ?
         BE    LDISP                * Yes -- go process
         CLI   VCARDF,C'S'          Set requested ?
         BNE   LINVFUNC             * No -- is invalid statement
```

Data-in-Virtual Sample Program Code (continued)

For a set request, the program determines whether the location to change does not extend past the maximum object size allowed. If the location is past the end of the current window, the program saves any existing changes to the object, and creates a window containing the page to be changed. It then changes the data in storage (but not in the linear data set).

For a display request, the program ensures the location to display is in the linear object (that is, the location is within the mapped area).

```
* SET: See if the location to change is within the range of the current
* MAP. If not, save any changes, get a larger area and issue a new MAP.
          C     R1,VSIZEB           Area to change within current MAP?
          BL    LGUPDCHR            * Yes -- continue
          C     R1,CSIZEMX          Area to change within max allowed?
          BNL   LINVADDR            * No -- is error
          CLI   VSWUPDT,0           Any updates to current MAP ?
          BE    LNOSVE1             * Yes -- then
          DIV   SAVE,ID=VTOKEN      Save any changes
          LA    R2,5                Error code from SAVE
          LTR   R15,R15             SAVE ok ?
          BNZ   LERROR              * No -- quit
          MVI   VSWUPDT,0           Clear update flag
LNOSVE1   L     R3,VSIZEB           Eliminate old map and storage
          LTR   R3,R3               Any to free ?
          BZ    LNOFREE             * Yes -- then
          DIV   UNMAP,ID=VTOKEN,AREA=VAREAPTR Release the MAP
          LA    R2,6                Error code from UNMAP
          LTR   R15,R15             UNMAP ok ?
          BNZ   LERROR              * No -- quit
          L     R1,VAREAPTR         R1 -> acquired storage
          FREEMAIN RU,A=(1),LV=(R3) Free the storage
LNOFREE   L     R2,VCHGADDR         Address of byte to change
          SRL   R2,12               R2 = page number - 1
          LA    R2,1(,R2)           R2 = page number to use
          ST    R2,VSIZEP           VSIZEP = MAP area in 4K units
          SLL   R2,12               R2 = size in bytes
          ST    R2,VSIZEB           VSIZEB = MAP area in bytes
          GETMAIN RU,LV=(R2),LOC=(ANY,ANY),BNDRY=PAGE get MAP area
          ST    R1,VAREAPTR         Save MAP area
          DIV   MAP,ID=VTOKEN,AREA=VAREAPTR,SPAN=VSIZEP
          LA    R2,3                Error code
          LTR   R15,R15             MAP work ok ?
          BNZ   LERROR              * No -- quit
LGUPDCHR  L     R1,VCHGADDR         R1 =  byte to change
          A     R1,VAREAPTR         R1 -> byte to change
          MVC   0(1,R1),VCARDV      Change the byte
          MVI   VSWUPDT,X'FF'       Show change made
          B     LGOODINP            Go print accept message
LDISP     EQU   *                   Display location contents
          L     R1,VCHGADDR         R1 =  location to display
          C     R1,VSIZEB           Ensure within current MAP
          BNL   LINVADDR            * If not, is error
          A     R1,VAREAPTR         R1 -> location to display
          MVC   VCARDV,0(R1)        Put into the card
```

Data-in-Virtual Sample Program Code (continued)

For both the set and display requests, the program displays the character at the specified location. For an invalid request, the program displays an error message. For all requests, the program then processes another statement.

When requested to terminate, the program saves any changes in the linear data set, terminates its use of the object (using UNIDENTIFY), and returns to the operating system.

```
LGOODINP EQU   *
         MVC   M1A,VCARDA          Address changed/displayed
         MVC   M1B,VCARDV          Storage value
         CLI   M1B,X'00'           If X'00' (untouched),
         BNE   LGOODIN1            * change to "?".
         MVI   M1B,C'?'            *
LGOODIN1 LA    R2,M1               R2 -> message to print
         B     LTELL               Go tell user status
LINVFUNC LA    R2,M2               Unknown function
         B     LTELL               Go tell user status
LINVADDV LA    R2,M3               Invalid address
         B     LTELL               Go tell user status
LINVADDR LA    R2,M4               Address out of range
LTELL    EQU   *                   R2 -> message to print
         LA    R4,L31B03           Return to L31B03 in 31-bit mode
         LA    R1,L24B03           Go to L24B03 in 24-bit mode
         BSM   R4,R1               R4 = A(X'80000000'+L31B03)
L24B03   PUT   VSYSPRT,(R2)        Print the message
         BSM   0,R4                Return to 31-bit mode at next instr
L31B03   B     LREAD               Continue
* End-of-file on SYSIN, or "E" function requested.
* Save any changes (DIV SAVE). Then issue UNIDENTIFY, which internally
* issues UNMAP and UNIDENTIFY.
LCLOSE   EQU   *
         CLI   VSWUPDT,0           Any updates outstanding ?
         BE    LCLOSE1             * No -- skip SAVE
         DIV   SAVE,ID=VTOKEN      Save any changes
         LA    R2,5                Error code from SAVE
         LTR   R15,R15             SAVE ok ?
         BNZ   LERROR              * No -- quit
LCLOSE1  DIV   UNIDENTIFY,ID=VTOKEN All done with object
         LA    R2,6                Error code from UNIDENTIFY
         LTR   R15,R15             UNIDENTIFY ok ?
         BNZ   LERROR              * No -- quit
         L     R13,4(,R13)         Unchain save areas and return
         LM    R14,R12,12(R13)     *
         SR    R15,R15             *
         BR    R14                 *
LERROR   ABEND (R2),DUMP           Take a dump
```

Data-in-Virtual Sample Program Code (continued)

These are the program's variables.

```
* Variables and constants for the program
VSVEAREA DC    18A(0)                Save area
VTOKEN   DC    XL8'00'               Object token
VAREAPTR DC    A(*-*)                -> MAP area
VSIZEP   DC    F'0'                  Size of MAP area, in pages (4K)
VSIZEB   DC    F'0'                  Size of MAP area, in bytes
VSWUPDT  DC    X'00'                 X'FF' -> map area updated
VCHGADDR DC    A(*-*),C' '           Address of byte to change/display
VTEMP8   DC    CL8' ',C' '           Temp area with buffer
VCARD    DC    CL80' '               Input card
VCARDF   EQU   VCARD+0,1             + Function (E/S/D)
VCARDA   EQU   VCARD+2,8             + Address to change/display
VCARDV   EQU   VCARD+11,1            + Character to change
CDIVDD   DC    X'5',C'DIVDD'         Linear Data Set DD pointer
* CTABTRT to verify string only has A thru F and 0 thru 9 (hex chars)
CTABTRT  DC    (C'A')X'FF',6X'00',(C'0'-C'F'-1)X'FF',10X'00',6X'FF'
* CTABTR & next line convert chars A:F,0:9 -> X'0A0B...0F000102...09'
CTABTR   EQU   *-C'A'
         DC    X'0A0B0C0D0E0F',(C'0'-C'F')X'00',X'010203040506070809'
CSIZEMX  DC    A(4096*1000)          Max size allowed for the DIV object
M1       DC    Y(M1E-*,0),C' Location '
M1A      DC    CL8' ',C' contains: '
M1B      DC    C' '
M1E      EQU   *
M2       DC    Y(M2E-*,0),C' Unknown function (not E/S/D)'
M2E      EQU   *
M3       DC    Y(M3E-*,0),C' Address not 8 hex characters'
M3E      EQU   *
M4       DC    Y(M4E-*,0),C' Address too big to set or display'
M4E      EQU   *
VSYSIN   DCB   MACRF=GM,DSORG=PS,RECFM=FB,LRECL=80,DDNAME=SYSIN,       *
               EODAD=LEOF
VSYSPRT  DCB   MACRF=PM,DSORG=PS,RECFM=VA,LRECL=133,DDNAME=SYSPRINT
R0       EQU   0                     Registers
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   ,
```

Executing the Program

The following JCL executes the program called DIVSAMPL. The function of DIVSAMPL is to change and display bytes (characters) in the data-in-virtual object, DIV.SAMPLE, that was allocated in "Creating a Linear Data Set" on page 82.

```
//DIV JOB .....
//DIV  EXEC PGM=DIVSAMPL
//STEPLIB DD DISP=SHR,DSN=DIV.LOAD
//DIVDD   DD DISP=OLD,DSN=DIV.SAMPLE
//SYSABEND DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
S 00001000 A  Changes byte X'1000' to "A"
D 00000F00    Displays "?" since byte X'F00' contains X'00'
S 00000F00 B  Changes byte X'F00' to "B"
S 00010000 C  Saves previous changes, gets new map, changes byte X'10000'
D 00001000    Displays "A" which was set by first statement
D 00000F00    Displays "B" which was set by third statement
E             Saves changes since last save (stmt 4), and terminates pgm
/*
```

DIVSAMPL reads statements from SYSIN that tell the program what to do. The format of each statement is **f aaaaaaaa v**, where:

f Function to perform:

- **S** Set a character in the object.
- **D** Display a character in the object.
- **E** End the program.

aaaaaaaa The hexadecimal address of the storage to set or display. Leading zeroes are required. The value must be less than X'003E8000'.

v For the S function, the character to put into the object.

Note: The program actually saves the change requested by the S function when either the user asks to change a byte past the current size of the object, or the user asks to terminate the program (E function).

Real Storage Management

The real storage manager (RSM) administers the use of real storage and directs the movement of virtual pages between auxiliary storage and real storage in page size (4096 bytes) blocks. It makes all addressable virtual storage in each address space appear as real storage. Only virtual pages necessary for program execution are kept in real storage, the remainder reside on auxiliary storage. RSM employs the auxiliary storage manager (ASM) of the Data Manager 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 I/O space on auxiliary storage. RSM relies on the system resources manager (SRM) for guidance in the performance of some of its operations.

RSM assigns storage page frames upon request from four pools of available frames, thereby associating virtual addresses with real storage addresses. Frames are repossessed upon termination of use, when freed by a user, when a user is swapped-out, or when needed to replenish an available pool. While a virtual page occupies a real storage frame, the page is considered pageable unless specified otherwise as a system page that must be resident in real storage. 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. Programs in this region do run in translation mode, although addressing is one to one virtual to real.

The paging services provided in MVS/XA include the following:

- PGRLSE or PGSER, RELEASE parameter - Release the virtual storage contents
- PGLOAD or PGSER, LOAD parameter - Load the virtual storage areas into real storage
- PGOUT or PGSER, OUT parameter - Page out the virtual storage areas from real storage

The page release function allows the user and the system to make available space in both real storage and auxiliary storage that is known to be of no future use. Proper use of this function can increase the amount of storage available to the system and prevent needless paging I/O activity. Usage of page release may improve operating efficiency when the using program can discard the contents of a large virtual storage area (circumscribing one or more pages) and reuse the virtual storage pages; paging operations may be eliminated for those virtual storage pages when they are reused.

The proper use of the page load and page out functions will tend to decrease system overhead resulting from page faults and to clean out of real storage those pages no longer required for program execution or not required for some period in the future.

Relinquishing Virtual Storage

When an area of virtual addressable storage within your program no longer has significant contents, you can make this storage available by issuing a PGRLSE macro instruction or by issuing the PGSER macro instruction with the RELEASE parameter specified. These macro instructions make available all real and external page storage wholly associated with the area of virtual address space specified. As shown in Figure 41, if the specified addresses are not on page boundaries, the low address is rounded up and the high address is rounded down; then, the pages contained between the addresses are released. The virtual space remains, but its contents are forfeited. When the using program can discard the contents of a large virtual area (one or more complete pages) and reuse the virtual space without the necessity of paging operations, the page release function may improve operating efficiency.

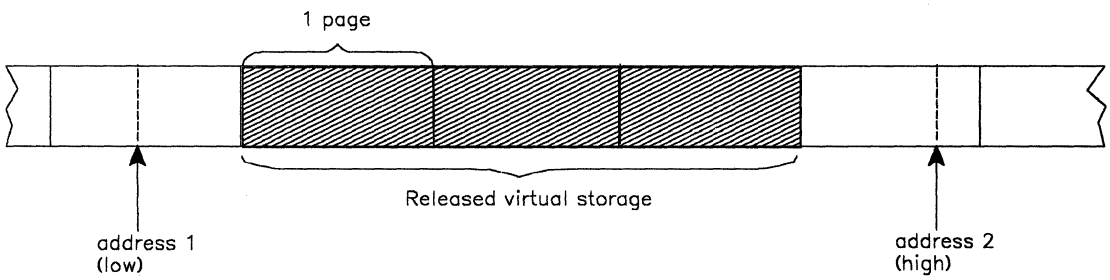

Figure 41. Releasing Virtual Storage

All storage obtained for your program by the GETMAIN macro instruction is automatically freed by the control program when the job step terminates. Freeing storage in this manner requires no action on your part. When you issue a FREEMAIN macro instruction, FREEMAIN does the equivalent of a page release for any resulting free page and the page is no longer available to the issuer.

Loading/Paging Out Virtual Storage Areas

The PGLOAD macro instruction and the PGSER macro with the LOAD parameter specified essentially provide a page-ahead function. By loading specified virtual storage areas into real storage, you can attempt to ensure that certain pages will be in real storage when needed. Page faults can occur, however, and these pages may be paged out.

With the page-load function, you have the option of specifying that the contents of the virtual area is to remain intact or be released. If you specify RELEASE=Y, the current contents of entire virtual 4K pages to be brought in may be discarded and new real frames assigned without page-in operations; if you specify RELEASE=N, the contents are to remain intact and be used later.

If you specify PGLOAD with RELEASE=Y or PGSER with LOAD and RELEASE=Y, the page release function will be performed before the page load function. That is, no page-in is needed for areas defining entire virtual pages since the contents of those pages are expendable.

The page-out function initiates page-out operations for specified virtual address pages that are in real storage. The real storage frames will be made available for reuse upon completion of the page-out operation unless you specify the KEEPREL parameter in the macro instruction.

An area that does not encompass one or more complete pages will be copied to auxiliary storage, but the real frames will not be freed.

Virtual Subarea List (VSL)

The virtual subarea list provides the basic input to the page service functions that use a 24-bit interface: PGLOAD, PGRLSE, and PGOUT. The list consists of one or more doubleword entries, each entry describing an area of virtual storage. The list must be nonpageable and contained in the address space of the subarea to be processed.

Each parameter list entry has the following format:

Byte	0	1	2	3	4	5	6	7
	FLAGS	START ADDRESS			FLAGS	END ADDRESS + 1		

Byte 0 Flags:

Bit 0	(1... ....)	This bit indicates that bytes 1-3 are a chain pointer to the next VSL entry to be processed; bytes 4-7 are ignored. This feature allows several parameter lists to be chained as a single logical parameter list.
Bit 1	(.1.. ....)	Reserved.
Bit 2	(..1. ....)	Reserved.
Bit 3	(...1 ....)	PGLOAD is to be performed; reserved, set by macro instruction.
Bit 4	(.... 1...)	PGRLSE is to be performed; reserved, set by macro instruction.
Bit 5	(.... .1..)	Reserved.
Bit 6	(.... ..1.)	Reserved.
Bit 7	(.... ...1)	Reserved.

Start Address:
The virtual address of the origin of the virtual area to be processed.

Byte 4 Flags:

Bit 0	(1... ....)	This flag indicates the last entry of the list. It is set in the last doubleword entry in the list.
Bit 1	(.1.. ....)	When this flag is set, the entry in which it is set is ignored.
Bit 2	(..1. ....)	Reserved.
Bit 3	(...1 ....)	This flag indicates that a return code of 4 was issued from a page service function other than PGRLSE.
Bit 4	(.... 1...)	Reserved.
Bit 5	(.... .1..)	PGOUT is to be performed; reserved, set by macro instruction.
Bit 6	(.... ..1.)	KEEPREL option of PGOUT is to be performed; reserved, set by macro instruction.
Bit 7	(.... ...1)	Reserved.

End Address + 1:
The virtual address of the byte immediately following the end of the virtual area.

Page Service List (PSL)

The page services list provides the basic input to the page service function for the PGSER macro instruction. You can specify either 24-bit or 31-bit addresses in the PSL entries. Each PSL entry specifies the range of addresses for which a service is to be performed or points to the first PSL entry in a new list of concatenated PSL entries that are to be processed. Within a PSL entry, you can also nullify a service on a range of addresses by indicating that you do not want to perform the service for that range.

Each 12-byte PSL entry has the following form:

Bytes	Meaning
0-3	Bit 0 of byte 0 must be 0. The remainder of these bytes contains the 31-bit starting address for which the page service is to be performed or a pointer to the next PSL.
4-7	Bit 0 of byte 4 must be 0. If bytes 0-3 contain the starting address, these bytes contain the address of the last byte for which the page service is to be performed. If bytes 0-3 contain a pointer to the next PSL, these bytes are reserved.
8	Flags set by the caller as follows:

Bit	Meaning
0	Set to 1 to indicate that this is the last PSL entry in a concatenation of PSL entries.
1	Set to 1 to indicate that no services are to be performed for the range of addresses specified.
2	Set to 1 to indicate that bytes 0-3 contain a pointer to the next PSL.

9-11	Set by the PGSER service routine.

Timing and Communication

This chapter describes timing services and communication services. Use communication services to send messages to the system operator, to TSO terminals, and to the system log. Use timing services to obtain the present date and time or for interval timing. Interval timing lets you set a time interval, test how much time is left in the interval, or cancel the interval.

Timing Services

Interval timing is a standard feature of MVS/XA. It provides the ability to request the date and time of day and provides for setting, testing, and canceling intervals of time.

Date and Time of Day

The operator is responsible for initially supplying the correct date and the time of day in terms of a 24-hour clock. You request the date and time of day using the TIME macro instruction. The control program returns the date in register 1 and the time of day in register 0 or in a doubleword that you supply if you specify the MIC or STCK parameter.

If ZONE=GMT is specified, the returned time of day and date will be for Greenwich Mean Time. If ZONE=LT is specified or if the ZONE parameter is omitted, the local time of day and date will be returned. However, if STCK is specified, the ZONE parameter will be ignored.

All references to time of day use the time-of-day (TOD) clock, a 64-bit binary counter. The TOD clock runs continuously while the power is on; the clock is not affected by the system stop-conditions. The operator normally sets the clock only after an interruption of CPU power has caused the clock to stop, and restoration of power has restarted it. The operator sets the clock during system initialization in response to a system message. (For more information about the TOD clock, see *Principles of Operation*.)

Interval Timing

Time intervals up to 24 hours in length can be established for any task in the job step through the use of the STIMER or STIMERM SET macro instructions. The time remaining in an interval established via the STIMER macro can be tested or cancelled through the use of TTIMER macro instruction. The time remaining in an interval established via the STIMERM SET macro instruction can be cancelled or tested through the use of the STIMERM CANCEL or STIMERM TEST macro instructions.

The value of the CPU timer can be obtained by using the CPUTIMER macro instruction. The CPU timer is used to track task-related time intervals.

The TASK, REAL, or WAIT parameters of the STIMER macro instruction and the WAIT=YES|NO parameter of the STIMERM SET macro instruction specify the manner in which the time interval is to be decreased. REAL and WAIT indicate the interval is to be decreased continuously, whether the associated task is active or not. TASK indicates the interval is to be decreased only when the associated task is active. STIMERM SET can establish real time intervals only.

If REAL or TASK is specified on STIMER or WAIT=NO is specified on STIMERM SET, the task continues to compete with the other ready tasks for control; if WAIT is specified on STIMER, or WAIT=YES is specified on STIMERM SET, the task is placed in a WAIT condition until the interval expires, at which time, the task is placed in the ready condition.

When TASK or REAL is specified on STIMER or WAIT=NO is specified on STIMERM SET, the address of an asynchronous timer completion exit routine can also be specified. This routine is given control sometime after the time interval completes. The delay is dependent on the system's work load and the relative dispatching priority of the associated task. If an exit routine is not specified, there is no notification of the completion of the time interval. The exit routine must be in virtual storage when specified, must save and restore registers as well as return control to the address in register 14.

Timing services does not serialize the use of asynchronous timer completion routines.

Figure 42 shows the use of a time interval when testing a new loop in a program. The STIMER macro instruction sets a time interval of 5.12 seconds, which is to be decreased only when the task is active, and provides the address of a routine called FIXUP to be given control when the time interval expires. The loop is controlled by a BXLE instruction.

```
         .
         .
         .
         STIMER TASK,FIXUP,BINTVL=TIME   Set time interval
LOOP     ...
         TM     TIMEXP,X'01'  Test if FIXUP routine entered
         BC     1,NG          Go out of loop if time interval expired
         BXLE   12,6,LOOP     If processing not complete, repeat loop
         TTIMER CANCEL        If loop completes, cancel remaining time
         .
         .
         .
NG       ...
         .
         .
         .
         USING  FIXUP,15      Provide addressability
FIXUP    SAVE   (14,12)       Save registers
         OI     TIMEXP,X'01'  Time interval expired, set switch in loop
         .
         .
         .
         RETURN (14,12)       Restore registers
         .
         .
         .
TIME     DC     X'00000200'   Timer is 5.12 seconds
TIMEXP   DC     X'00'         Timer switch
```

Figure 42. Interval Processing

The loop continues as long as the value in register 12 is less than or equal to the value in register 6. If the loop stops, the TTIMER macro instruction causes any time remaining in the interval to be canceled; the exit routine is not given control. If, however, the loop is still in effect when the time interval expires, control is given to the exit routine FIXUP. The exit routine saves registers and turns on the switch tested in the loop. The FIXUP routine could also print out a message indicating that the loop did not go to completion. Registers are restored and control is returned to the control program. The control program returns control to the main program and execution continues. When the switch is tested this time, the branch is taken out of the loop. Caution should be used to prevent a timer exit routine from issuing an STIMER specifying the same exit routine. An infinite loop may occur.

The priorities of other tasks in the system may also affect the accuracy of the time interval measurement. If you code REAL or WAIT, the interval is decreased continuously and may expire when the task is not active. (This is certain to happen when WAIT is coded.) After the time interval expires, assuming the task is not in the wait condition for any other reason, the task is placed in the ready condition and then competes for CPU time with the other tasks in the system that are also in the ready condition. The additional time required before the task becomes active will then depend on the relative dispatching priority of the task.

The STIMER macro instruction should not be issued while a BTAM OPEN or LINE OPEN operation is in progress, since the BTAM OPEN LINE routines also use STIMER. STIMER should not be issued before invoking dynamic allocation because dynamic allocation can also issue STIMER.

Communicating with the System Operator

The WTO and the WTOR macro instructions allow you to write messages to the operator. The WTOR macro instruction also allows you to request a reply from the operator. Messages can be sent to (and replies received from) as many as 99 operator consoles. Only standard, printable EBCDIC characters, shown in Figure 43, appear on the console. All other characters are replaced by blanks. If the terminal does not have dual-case capability, it prints lowercase characters as uppercase characters.

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

Figure 43. Characters Printed or Displayed on an MCS Console

Notes:

1. *If the display device or printer is defined to JES3, the following characters are also translated to blanks:*
   | ! ; ¬ : ″

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

There are two basic forms of the WTO macro instruction: the single-line form, and the multiple-line form.

The following should be considered when issuing multiple-line WTO messages (MLWTO).

- Only the first line of a multiple-line WTO message is passed to the installation-written WTO exit routine (IEECVXIT).

- When a console switch takes place, unended multiple-line WTO messages and multiple-line WTO messages in the process of being written to the original console are not moved to the new console.

- When a hard copy switch takes place from the system log to an active operator's console, MLWTO messages in the process of being written to the system log are not moved to the new hard copy device.

- The left-most three bytes of register zero must be zero for a multiple-line message. You must ensure that this is done.

- When the system hard copy log is an active operator's console, only the hard copy versions of multiple-line messages are written to the console.

- Because the hard copy log receives a copy of every message in the system, use an active operator's console as the hard copy log only in an emergency.

See the macro instructions section for an explanation of the parameters in the single-line and multiple-line forms of the WTO macro instruction.

The message is routed using the routing codes specified in the WTO macro instruction. At system generation, each operator's console in the system is assigned routing codes that correspond to the functions that the installation wants that console to perform. When any of the routing codes assigned to a message match any of the routing codes assigned to a console, the message is sent to that console.

Disposition of the message is indicated through the descriptor codes specified in the WTO macro instruction. Descriptor codes classify WTO messages so that they can be properly presented on, and deleted from, display devices. Each WTO macro instruction should contain at least one descriptor code. The descriptor code is not printed or displayed as part of the message text.

If the user supplies a descriptor code in the WTO macro instruction, an indicator is inserted at the start of the message. The indicators are: a blank, an at sign (@), an asterisk (*), or a blank followed by a plus sign (+). The indicator inserted in the message depends on the descriptor code that the user supplies and whether the user is a privileged or APF-authorized program or a non-authorized problem program. Figure 44 shows the indicator that is used for each descriptor code.

Descriptor Code	Privileged or APF-Authorized Program	Non-Authorized Problem Program
1	*	@
2	*	@
3-10	blank	blank +
11	*	@
12-16	blank	blank +

Figure 44. Descriptor Code Indicators

The indicator @ or * informs operators that they must take some immediate or critical eventual action. A critical eventual action is an action that the operator must perform, as soon as possible, in response to a critical situation during the operation of the system. For example, if the dump data set is full, the operator is notified to mount a new tape on a specific unit. This is considered a critical action because no dumps can be taken until the tape is mounted; it is eventual rather than immediate because the system continues to run and processes jobs that do not require dumps.

Action messages to the operator, which are identified by the @ or * indicator, can be individually suppressed by the installation. When a program invokes WTO or WTOR to send a message, the system determines if the message is to be suppressed. If it is, the system writes the message to the hardcopy log and the operator does not receive it on the screen.

If a problem program issues a message with descriptor code of 1 or 2, descriptor code 7 is forced so that the message is deleted at address space or task termination. For more information concerning routing and descriptor codes, see *Routing and Descriptor Codes.*

If an application that uses WTO needs to alter a message each time the message is issued, the list form of the WTO macro may be useful. The message area, which is referenced by the WTO parameter list, can be altered before you issue the WTO. The message length, which appears in the WTO parameter list, does not need to be altered if you pad out the message area with blanks.

A sample WTO macro instruction is shown in Figure 45.

```
Single-line WTO   'BREAKOFF POINT REACHED.  TRACKING COMPLETED.', C
format             ROUTCDE=14,DESC=7

Multiple-   WTO    ('SUBROUTINES CALLED',C),                        C
line format        ('ROUTINE TIMES CALLED',L),('SUBQUER',D),        C
(list form)        ('ENQUER',D),('WRITER',D),                       C
                   ('DQUER',DE),                                    C
                   ROUTCDE=(2,14),DESC=(7,8),MF=L
```

Figure 45. Writing to the Operator

To use the WTOR macro instruction, code the message exactly as designated in the single-line WTO macro instruction. (The WTOR macro instruction cannot be used to pass multiple-line messages.) When the message is written, the control program adds a two-character message identifier before the message to associate the reply with the message. The control program also inserts an indicator as the first character of all WTOR messages, thereby informing the operator that immediate action is required. You must, however, indicate the response desired. In addition, you must supply the address of the area in which the control program is to place the reply, and you must indicate the length of the reply. The length of the reply may not be zero. You also supply the address of an event control block which the control program posts after the reply has been placed, left-adjusted, in your designated area.

A sample WTOR macro instruction is shown in Figure 46. The reply is not necessarily available at the address you specified until a WAIT macro instruction has been issued.

```
            .
            .
            .
            XC      ECBAD,ECBAD Clear ECB
            WTOR    'STANDARD OPERATING CONDITIONS?  REPLY YES OR NO', C
                    REPLY,3,ECBAD,ROUTCDE=(1,15)
            WAIT    ECB=ECBAD
            .
            .
            .
ECBAD       DC      F'0'            Event control block
REPLY       DC      C'bbb'          Answer area
```

Figure 46. Writing to the Operator With a Reply

When a WTOR macro instruction is issued, any console receiving the message has the authority to reply. The first reply received by the control program is returned to the issuer of the WTOR, providing the syntax of the reply is correct. If the syntax of the reply is not correct, another reply is accepted. The WTOR is satisfied when the control program moves the reply into the issuer's reply area and posts the event control block. Each console that received the original WTOR will also receive the accepted reply unless it's a security message. No console receives the accepted reply to a security message. The master console may answer any WTOR, even if it did not receive the original message.

Writing to the Programmer

The WTO and the WTOR macro instructions allow you to write messages to the programmer, as well as to the operator. However, only the operator can reply to a WTOR message.

To write a message to the programmer, you must specify ROUTCDE=11 in the WTO or the WTOR macro instruction.

Writing to the System Log

The system log consists of one SYSOUT data set on which the communication between the operator and the system is recorded. You can use the system log by coding the information that you wish to log in the "text" parameter of the WTL macro instruction.

When the WTL macro instruction is executed, the control program places your text in one of the buffers and, when the buffer is full, writes the buffer onto the system log data set. The control program writes the text of your WTL macro instruction on the master console instead of on the system log if the system log is not active.

Although when using the WTL macro instruction you code the message within apostrophes, the written message does not contain the apostrophes. The message can include any character that is valid for the WTO macro instruction and is assembled and written the same way as the WTO macro instruction. MCS routing codes and descriptor codes are not assigned, since they are not needed by the WTL macro instruction.

Note: The exact format of the output of the WTL macro instruction varies depending on the job entry system (JES2 or JES3) that is being used, the output class that is assigned to the log at system initialization, and whether DLOG is in effect for JES3. In JES3, system log entries are preceded by a 23-character prefix
that includes a time stamp and routing information. If the combined prefix and message exceeds 126 characters, the log entry is split at the first blank or comma encountered when scanning backward from the 126th character of the combined prefix and message. See *Operations: JES3 Commands* for information about the format of the log entry when using JES3.

Message Deletion

When using a display console, messages that are no longer necessary can be deleted from the operator's screen by the programmer. The control program assigns a message identification number to each WTO and WTOR message and returns the message identification number in register 1. The DOM macro instruction uses the identification number to indicate which message is to be deleted. The message identification number must not be confused with the reply identification number that is assigned to WTOR replies.

Deleting messages is simplified by the TOKEN parameter of the DOM, WTO, and WTOR macros. TOKEN identifies a unique 4-byte ID that you originate and associate with one or more messages when you issue a WTO or WTOR. Then you can use the same token ID when you issue a DOM macro to delete all the messages that are associated with the ID.

When you want to delete several (up to 60) messages with a single DOM invocation, you can use the COUNT parameter with MSGLIST. Count is used to indicate the number of messages in the message list. The high order bit of each message ID in the message list must be zero.

You can also use the DOM macro instruction to keep WTOR messages from appearing, or erase them if they have already appeared, by specifying REPLY=YES on the macro. The issuer of the DOM with REPLY=YES must be a task in the same job step and address space as the issuer of the WTOR macro instruction or must be a task executing in supervisor state, in key 0-7, or authorized by APF.

Because all outstanding WTOs that were issued with a descriptor code of 7 are deleted at address space or task termination, it is possible for a WTO to be deleted without ever being displayed. If a task terminates and the JES global processor is not active or if the console to which the message is routed is backed up, the message is deleted.

Part II: Macro Instructions

You can communicate service requests to the control program using a set of macro instructions provided by IBM. These macro instructions are available only when programming in the assembler language, and are processed by the assembler program using macro definitions supplied by IBM and placed in the macro library when the system was generated.

The processing of the macro instruction by the assembler program results in a macro expansion, generally consisting of data and executable instructions in the form of assembler language statements. The data fields are the parameters to be passed to the requested control program routine; the executable instructions generally consist of a branch around the data, instructions to load registers, and either a branch instruction or a supervisor call (SVC) to give control to the proper program. The exact macro expansion appears as part of the assembler output listing.

Applications programmers can use the macro instructions described in this publication without restriction. Some macro instructions contain parameters that are restricted to systems programmers and installation-approved personnel. These parameters, as well as installation-controlled macro instructions, are described in *SPL: System Macros and Facilities.*

Selecting the Macro Level

Certain MVS/XA macro expansions cannot execute on an MVS/370 system. These macros are not downwardly compatible. Macros that are new for MVS/XA or new parameters are not supported by the MVS/370 versions of the downward incompatible macros. In some cases the new parameters are ignored, in other cases they cause assembly errors. Callers executing in 31-bit addressing mode must use the MVS/XA version of these downward incompatible macro instructions. The following macro instructions are the downward incompatible macros described in this book:

- ATTACH
- ESTAE
- EVENTS
- STIMER
- WTOR

The SPLEVEL macro instruction solves the problems associated with downward incompatible macros. The SPLEVEL macro instruction allows an installation to assemble programs using the MVS/XA macro library and to select either the MVS/370 or the MVS/XA version of the downward incompatible macros.

Before issuing a downward incompatible macro, users can specify the macro level that they want. They do this by issuing the SPLEVEL macro using the SET=*n* option, with *n*=1 or 2. If *n*=1, the MVS/370 expansion of the macro code is generated and if *n*=2, the MVS/XA expansion of the macro code is generated. If the user does not specify the value of *n*, the SPLEVEL routine uses the default value of 2. See *SPL: System Modifications* for information concerning the way in which an installation can change this default.

A user can also select the level of the macro at execution time, based on the system that is operating. The example in Figure 47 shows one method of selecting the macro level. The example uses the WTOR macro instruction, but any downward incompatible macro instruction could be substituted. The code makes use of the fact that the CVTMVSE bit in byte CVTDCB (located at offset 116 or X'74' of the communications vector table (CVT)) is set to 1 when System Product Version 2 is operating. The CVTMVSE field of the CVT is defined in System Product Version 2.

```
*     DETERMINE WHICH SYSTEM IS EXECUTING
          TM        CVTDCB,CVTMVSE
          BO        SP2
*     INVOKE MVS/370 VERSION OF THE MACRO
          SPLEVEL   SET=1
          WTOR      ...
          B         CONTINUE
*     INVOKE MVS/XA VERSION OF THE MACRO
SP2       SPLEVEL   SET=2
          WTOR      ...
*     RESET TO SYSTEM DEFAULT
CONTINUE  SPLEVEL   SET
```

Figure 47. Macro Level Selected at Execution Time

The SPLEVEL macro instruction is also described in *SPL: System Macros and Facilities.*

Addressing Mode and the Macro Instructions

Callers in either 24-bit or 31-bit addressing mode can invoke most of the macros described in this book. The following is a list of the macro instructions, documented in Part II of this book, that require the caller to be executing in 24-bit addressing mode and require that the parameters be located in 24-bit addressable storage:

- SEGLD
- SEGWT
- SPIE

All addresses specified as parameters for the other macro instructions in this book can be 31-bit addresses unless otherwise stated. If a parameter passed by a program executing in 31-bit addressing mode must be located in 24-bit addressable storage, the restriction is stated in the description of the macro instruction.

In general, a program executing in 24-bit addressing mode cannot pass parameters located above 16 Mb in virtual storage to a system service. There are exceptions to this general rule. For example, a program executing in 24-bit addressing mode can:

- Free storage above 16 Mb using the FREEMAIN macro instruction
- Allocate storage above 16 Mb using the GETMAIN macro instruction
- Perform cell pool services for cell pools located in storage above 16 Mb using the CPOOL macro instruction
- Perform page services for storage locations above 16 Mb using the PGSER macro instruction

See the descriptions of the individual macro instructions for details.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of these macro instructions:

- ATTACH
- CALL
- ESTAE
- EVENTS
- LINK
- SETRP
- STIMER
- SYNCH
- WTOR
- XCTL

Macro Instruction Forms

When written in the standard form, some of the macro instructions result in instructions that store into an inline parameter list. The option of storing into an out-of-line parameter list is provided to allow the use of these macro instructions in a reenterable program. You can request this option through the use of list and execute forms. When list and execute forms exist for a macro instruction, their descriptions follow the description of the standard form.

Use the list form of the macro instruction to provide a parameter list to be passed either to the control program or to a problem program, depending on the macro instruction. The expansion of the list form contains no executable instructions; therefore registers cannot be used in the list form.

Use the execute form of the macro instruction in conjunction with one or two parameter lists established using the list form. The expansion of the execute form provides the executable instructions required to modify the parameter lists and to pass control to the required program. Only the ATTACH, LINK, and XCTL macro instructions use two parameter lists: a problem program list, resulting from the address parameter and VL parameters, and a control program list, resulting from the remaining parameters. The control program list is required, and the problem program list is optional in these macro instructions.

If you do not generate the control program parameter list form of the macro, you must provide the list yourself, initialize it, then update it (either directly or by explicitly specifying keywords on the execute form).

Note: If the program issuing the execute form of a macro instruction is executing in 24-bit addressing mode, the remote control program parameter list must be located in 24-bit addressable storage.

The CALL, DEQ, ENQ, and SNAP macro instructions can result in variable length parameter lists. The length of the parameter list generated by the list form of the macro instruction must be equal to the maximum length list required by any execute form that refers to the list. The maximum length list can be constructed in one of three methods:

- Code the parameters required for the maximum length execute form in the list form.
- Provide a DS instruction immediately following the list form to allow for the maximum length parameter list.
- Acquire a maximum length list by using commas in the list form to indicate the maximum number of parameters. For example, the STORAGE parameter of the SNAP macro instruction could be coded as STORAGE=(,,,,,,,,) to allow for five pairs of addresses. The actual addresses would be provided in the execute forms.

The descriptions of the following macro instructions assume that the standard begin, end, and continue columns are used - for example, column 1 is assumed as the begin column. To change the begin, end, and continue columns, code the ICTL instruction to establish the coding format you wish to use. If you do not use ICTL, the assembler recognizes the standard columns. To code the ICTL instruction, see *Assembler H Version 2 Application Programming: Language Reference.*

Coding the Macro Instructions

The table appearing near the beginning of each macro instruction indicates how the macro instruction is to be coded. The table does not explain the meanings of the parameters; the parameters are explained following the table.

Figure 48 shows a sample macro instruction, TEST, and summarizes all the coding information that is available for it. The table is divided into three columns, **A**, **B**, and **C**.

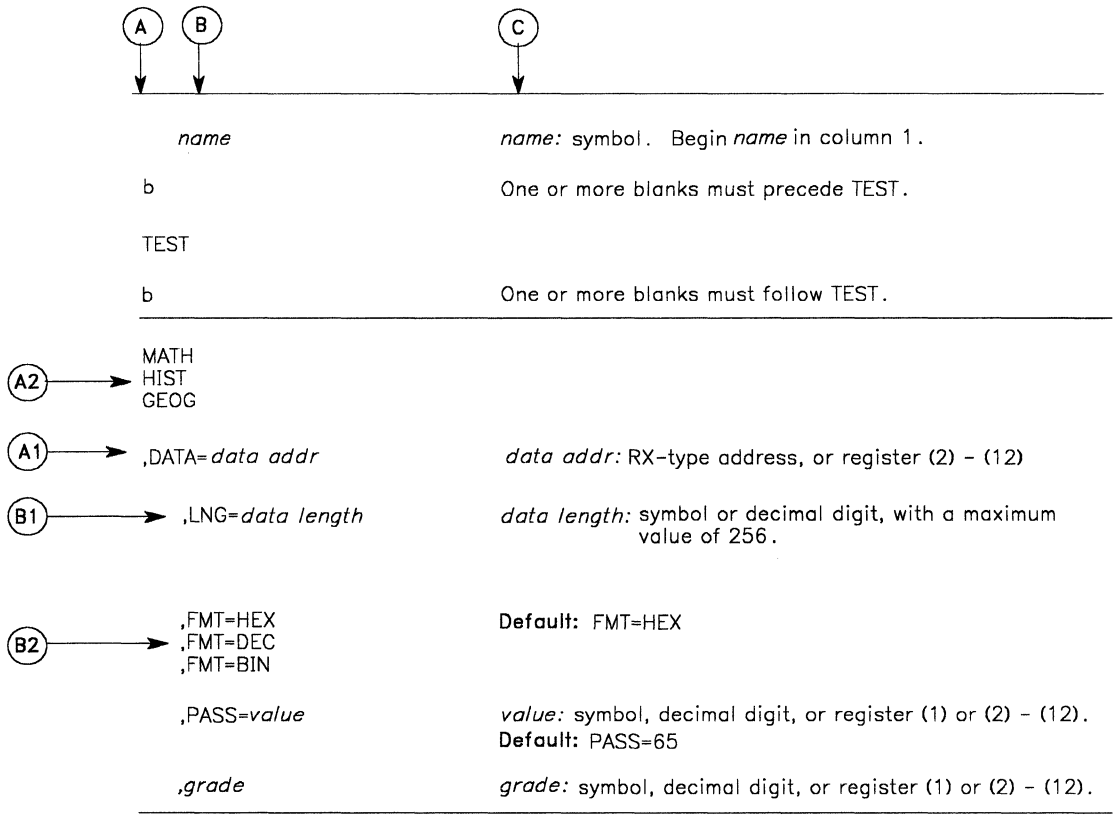

Figure 48. Sample Macro Instruction

- The first column, **A**, contains those parameters that are required for that macro instruction. If a single line appears in that column, **A1**, the parameter on that line is required and must be coded. If two or more lines appear together, **A2**, one and only one of the parameters on the lines must be coded.

- The second column, **B**, contains those parameters that are optional for that macro instruction. If a single line appears in that column, **B1**, the parameter on that line is optional. If two or more lines appear together, **B2**, one and only one of the parameters appearing on lines may be coded if desired.

- The third column, **C**, provides additional information for coding the macro instruction. The following terms can appear in column **C**.

symbol: any symbol valid in the assembler language. That is, an alphabetic character followed by 0-7 alphameric characters, with no special characters and no blanks.

decimal digit: any decimal digit up to the value indicated in the parameter description. If both symbol and decimal digit are indicated, an absolute expression is also allowed.

register (2) - (12): one of general registers 2 through 12, specified within parentheses, previously loaded with the right-adjusted value or address indicated in the parameter description. The unused high-order bits must be set to zero. The register may be designated symbolically or with an absolute expression.

register (0): general register 0, previously loaded as indicated under register (2) - (12) above. Designate the register as (0) only.

register (1): general register 1, previously loaded as indicated under register (2) - (12) above. Designate the register as (1) only.

RX-type address: any address that is valid in an RX-type instruction (for example, LA).

A-type address: any address that may be written in an A-type address constant.

default: a value that is used in default of a specified value, and that is assumed if the parameter is not coded.

Use the parameters to specify the services and options to be performed, and write them according to the following general rules:

- If the selected parameter is written in all capital letters (for example, MATH, HIST, or FMT=HEX), code the parameter exactly as shown.
- If the selected parameter is written in italics (for example, *grade*) substitute the indicated value, address, or name.
- If the selected parameter is a combination of capital letters and italics separated by an equal sign (for example, DATA=*data addr*) code the capital letters and equal sign as shown, and then make the indicated substitution for the italics.
- Read the table from top to bottom, and code the parameters in the order shown. Code commas and parentheses exactly as shown.
- If you select a parameter to be coded, read the third column, **C**, before proceeding to the next parameter. Column **C** often contains notes pertaining to restrictions on coding the parameters.

Continuation Lines

You can continue the parameter field of a macro instruction on one or more additional lines according to the following rules:

1. Enter a continuation character (not blank, and not part of the parameter coding) in column 72 of the line.

2. Continue the parameter field on the next line, starting in column 16. All columns to the left of column 16 must be blank.

You can code the parameter field being continued in one of two ways. Code the parameter field through column 71, with no blanks, and continue in column 16 of the next line; or truncate the parameter field by a comma, where a comma normally falls, with at least one blank before column 71, and then continue in column 16 of the next line. Figure 49 shows an example of each method. Additional information on the continuation of any assembler language macro instruction is provided in the publication *Assembler Language*.

```
1         10        16                                  44                                  72
↓         ↓         ↓                                   ↓                                   ↓
NAME1     OP1       OPERAND1,OPERAND2,OPERAND3,OPERAND4,OPERAND5,OPERAND6,OPX
                    ERAND7                              THIS IS ONE WAY
NAME2     OP2       OPERAND1,OPERAND2,                  THIS IS ANOTHER WAY                 X
                    OPERAND3,OPERAND4,                  ANOTHER                             x
                    OPERAND5,OPERAND6,OPERAND7          WAY
```

Figure 49. Continuation Coding

ABEND — Abnormally Terminate a Task

The ABEND macro instruction initiates error processing for a task. ABEND can request a full or tailored dump of virtual storage areas and control blocks pertaining to the tasks being abnormally terminated, and can specify that the entire job step is to be abnormally terminated. Before the task is terminated, an ESTAE exit gets control. This exit may recover the task and allow it to retry.

If the job step task is abnormally terminated or if ABEND specifies job step termination, the completion code is recorded on the system output device, and the remaining job steps in the job are either skipped or executed as specified in their job control statements.

If the job step is not to be terminated, the system takes the following actions:

- It terminates the task that was active when ABEND was issued, along with all of the subtasks of that active task.
- It posts the completion code as indicated in the completion code parameter description below.
- It gives control to the end-of-task exit routine specified in the ATTACH macro instruction that created the task which issued ABEND. The exit routine is given control when the originating task of the task for which ABEND was issued becomes active. None of the end-of-task exit routines specified for any subtasks of the task for which ABEND was issued are given control.

The ABEND macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ABEND.
ABEND	
b	One or more blanks must follow ABEND.
comp code	*comp code:* symbol, decimal or hexadecimal digit, or register (1) or (2) - (12). **Value range:** 0 - 4095
,REASON=*reason code*	*reason code:* symbol, decimal or hexadecimal number, or register (2) - (12).
,DUMP ,,STEP ,,,*code type* ,DUMP,STEP ,DUMP,,*code type* ,,STEP,*code type* ,DUMP,STEP,*code type*	*code type:* USER or SYSTEM. **Default:** *code type* = USER.
,DUMPOPT=*parm list addr*	*parm list addr:* RX-type address, or register (2) - (12).

The parameters are explained as follows:

comp code
specifies the completion code associated with the abnormal termination. If the job step is to be terminated, the decimal representation of the user completion code or the hexadecimal representation of the system completion code is recorded on the system output device. If the job step is not to be terminated, the completion code is placed in the

TCB of the active task, and in the ECB specified in the ECB parameter of the ATTACH macro instruction issued to create the active task. If you specify a hexadecimal digit, you must use X'dd' format to distinguish the hexadecimal from decimal.

,REASON=*reason code*
specifies the *reason code* that the user wants to pass to subsequent recovery exits. The value range for the *reason code* is a 32-bit hexadecimal number or a 31-bit decimal number. This *reason code* supplements the completion code associated with an abnormal termination, allowing the user to uniquely identify the cause of the abnormal termination. The recovery termination manager propagates the *reason code* to each recovery exit and to the TCB and ASCB control blocks, making it available for system messages.

,DUMP
,,STEP
,,,*code type*
,DUMP,STEP
,DUMP,,*code type*
,,STEP,*code type*
,DUMP,STEP,*code type*
specifies options available with the ABEND macro instruction:

DUMP specifies that a dump is requested of virtual storage areas assigned to the task and control blocks pertaining to the task. A separate dump is provided for each of the tasks being terminated as a result of ABEND. If a //SYSABEND, //SYSMDUMP, or //SYSUDUMP DD statement is not provided, the DUMP parameter is ignored.

STEP specifies that the entire job step of the active task is to be abnormally terminated.

Note: If the STEP parameter is coded in an ABEND macro under TSO, the TSO job will be terminated.

code type specifies that the completion code is to be treated as a USER or SYSTEM code.

,DUMPOPT=*parm list addr*
specifies the address of a parameter list valid for the SNAP macro instruction. The parameter list is used to produce a tailored dump, and may be created by using the list form of the SNAP macro instruction, or a compatible list may be created. The TCB, DCB, ID, and STRHDR options available on SNAP will be ignored if they appear in the parameter list; the TCB used will be that of the task being terminated, the DCB used will be provided by the ABDUMP routine. If a //SYSABEND, //SYSMDUMP, or //SYSUDUMP DD statement is not provided, the DUMPOPT parameter is ignored.

If the dump options specified include ranges of storage areas to be dumped, only the storage areas in the first thirty ranges will be dumped. If SUBPLST is specified in the SNAP parameter list passed to the ABEND macro instruction via DUMPOPT, the first seven subpools will be dumped.

Example 1

Operation: Terminate with a user completion code of 432.

```
ABEND 432
```

Example 2

Operation: Terminate with the user completion code that is contained in register 5. The entire job step is to be terminated.

```
ABEND (5),,STEP
```

Example 3

Operation: Terminate with a system completion code of X'0C4'.

```
ABEND X'0C4',,,SYSTEM
```

ATTACH — Create a New Task

The ATTACH macro instruction creates a new task and indicates the entry point in the program to be given control when the new task becomes active. The entry point name that is specified must be a member name or an alias in a directory of a partitioned data set, or must have been specified in an IDENTIFY macro instruction. If the specified entry point cannot be located, the new subtask is abnormally terminated.

On entry to the attached routine, the high-order bit, bit 0, of register 14 is set to indicate the addressing mode of the issuer of the ATTACH macro. If bit 0 is 0, the issuer is executing in 24-bit addressing mode; if bit 0 is 1, the issuer is executing in 31-bit addressing mode.

The address of the task control block for the new task is returned in register 1. The new task is a subtask of the originating task; the originating task is the task that was active when the ATTACH macro instruction was issued. The limit and dispatching priorities of the new task are the same as those of the originating task unless modified in the ATTACH macro instruction.

The load module containing the program to be given control is brought into virtual storage if a usable copy is not available in virtual storage. The issuing program can provide an event control block, in which termination of the new task is posted, an exit routine to be given control when the new task is terminated, and a parameter list whose address is passed in register 1 to the new task. If the ECB or ETXR parameter is coded, a DETACH macro instruction must be issued to remove the subtask from the system before the program that issued the ATTACH macro instruction terminates. If the ECB or ETXR parameter is not coded, the subtask is automatically removed from the system upon completion of its execution. If the ECB parameter is specified in the ATTACH macro instruction, the ECB must be in storage so that the issuer of the attach can wait on it (using the WAIT macro instruction) and the control program can post it on behalf of the terminating task. The ATTACH macro instruction can also be used to specify that ownership of virtual subpools is to be assigned to the new task, or that the subpools are to be shared by the originating task and the new task.

This macro can be assembled compatibly between MVS/XA and MVS/370 through the use of the SPLEVEL macro instruction. Default processing will result in an expansion of the macro that operates only with MVS/XA. See the topic "Selecting the Macro Level" for additional information. If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction. Except for the address of the DCB, all input parameters to the ATTACH macro instruction can have addresses greater than 16 Mb if the issuer is executing in 31-bit addressing mode.

The standard form of the ATTACH macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ATTACH.
ATTACH	
b	One or more blanks must follow ATTACH.

EP=*entry name* EPLOC=*entry name addr* DE=*list entry addr*	*entry name:* symbol. *entry name addr:* A-type address, or register (2) - (12). *list entry addr:* A-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* A-type address, or register (2) - (12).
,LPMOD=*limit prior nmbr*	*limit prior nmbr:* symbol, decimal digit, or register (2) - (12).
,DPMOD=*disp prior nmbr*	*disp prior nmbr:* symbol, decimal digit, or register (2) - (12).
,PARAM=(*addr*) ,PARAM=(*addr*),VL=1	*addr:* A-type address, or register (2) - (12). **Note:** *addr* is one or more addresses, separated by commas. For example, PARAM=(*addr,addr,addr*)
,ECB=*ecb addr*	*ecb addr:* A-type address, or register (2) - (12).
,ETXR=*exit rtn addr*	*exit rtn addr:* A-type address, or register (2) - (12).
,GSPV=*subpool nmbr* ,GSPL=*subpool list addr*	*subpool nmbr:* symbol, decimal digit, or register (2) - (12). *subpool list addr:* A-type address, or register (2) - (12).
,SHSPV=*subpool nmbr* ,SHSPL=*subpool list addr*	*subpool nmbr:* symbol, decimal digit, or register (2)-(12). *subpool list addr:* A-type address, or register (2)-(12).
,SZERO=YES ,SZERO=NO	**Default:** SZERO=YES
,TASKLIB=*dcb addr*	*dcb addr:* A-type address, or register (2)-(12).
,STAI=(*exit addr*) ,STAI=(*exit addr,parm addr*) ,ESTAI=(*exit addr*) ,ESTAI=(*exit addr,parm addr*)	*exit addr:* A-type address, or register (2)-(12). *parm addr:* A-type address, or register (2)-(12).
,PURGE=QUIESCE ,PURGE=NONE ,PURGE=HALT	**Note:** PURGE may be specified only if STAI or ESTAI is specified. **Default** for STAI: PURGE=QUIESCE **Default** for ESTAI: PURGE=NONE
,ASYNCH=NO ,ASYNCH=YES	**Note:** ASYNCH may be specified only if STAI or ESTAI is specified. **Default** for STAI: ASYNCH=NO **Default** for ESTAI: ASYNCH=YES
,TERM=NO ,TERM=YES	**Note:** TERM may be specified only if ESTAI is specified. **Default:** TERM=NO
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

EP=*entry name*
EPLOC=*entry name addr*
DE=*list entry addr*

specifies the entry name, the address of the entry name, or the address of the name field of a 60-byte list entry for the entry name that was constructed using the BLDL macro instruction. If EPLOC is coded, the name must be padded to eight bytes, if necessary.

If an unauthorized program issues the ATTACH macro instruction and the DE parameter specifies an entry in an authorized library, the program-supplied DE information is ignored for integrity reasons. Instead, contents management uses the BLDL macro instruction to construct a new list entry containing the DE information for the ATTACH. The DE information supplied by an unauthorized program will also be ignored if the

ATTACH macro instruction is requesting access to a program or library that is controlled by the System Authorization Facility.

When you use the DE parameter with the ATTACH macro, DE specifies the address of a list that was created by a BLDL macro. BLDL and ATTACH must be issued from the same task; otherwise, the system might terminate the program with an abend code of 106 and a return code of 15. Therefore, do not issue an ATTACH or DETACH between issuances of BLDL and ATTACH for the module.

,DCB=*dcb addr*
specifies the address of the data control block for the partitioned data set containing the entry name described above. (Note: The DCB must be opened before the ATTACH macro instruction is executed and must be the DCB used in the BLDL that built the 60 byte DE list entry. The DCB must remain open until the subtask becomes active, and it should not be closed immediately following the attach macro.)

Note: DCB must reside in 24-bit addressable storage.

,LPMOD=*limit prior nmbr*
specifies the number (255 or less) to be subtracted from the current limit priority of the originating task. The result is the limit priority of the new task. If this parameter is omitted, the current limit priority of the originating task is assigned as the limit priority of the new task.

,DPMOD=*disp prior nmbr*
specifies the signed number (255 or less) to be algebraically added to the current dispatching priority of the originating task. The result is assigned as the dispatching priority of the new task, unless it is greater than the limit priority of the new task. If the result is greater, the limit priority is assigned as the dispatching priority.

If a register is designated, a negative number must be in two's complement form in the register. If this parameter is omitted, the dispatching priority assigned is the smaller of either the new task's limit priority or the originating task's dispatching priority.

,PARAM=(*addr***)**
,PARAM=(*addr***),VL=1**
specifies address(es) to be passed to the control program. Each address is expanded inline to a fullword on a fullword boundary, in the order designated. Register 1 contains the address of the first word when the program is given control.

VL=1 should be designated only if the called program can be passed a variable number of parameters. VL=1 causes the high-order bit of the last address to be set to 1; the bit can be checked to find the end of the list.

,ECB=*ecb addr*
specifies the address of an event control block for the new task to be used by the control program to indicate the termination of the new task. The ECB must be in storage so that the issuer of the attach can wait on it (using the WAIT macro instruction) and the control program can post it on behalf of the terminating task. The return code (if the task is terminated normally) or the completion code (if the task is terminated abnormally) is also placed in the event control block. If this parameter is coded, a DETACH macro instruction must be issued to remove the subtask from the system after the subtask has been terminated.

,ETXR = *exit rtn addr*

specifies the address of the end-of-task exit routine to be given control after the new task is normally or abnormally terminated. The exit routine is given control when the originating task becomes active after the subtask is terminated, and must be in virtual storage when required. If this parameter is coded, a DETACH macro instruction must be issued to remove the subtask from the system after the subtask has been terminated.

The exit routine receives control in the addressing mode of the issuer of the ATTACH macro instruction. ATTACH processing issues an ABEND with completion code X'72A' if a task attempts to create two subtasks with the same exit routine in different addressing modes.

The contents of the registers when the exit routine is given control are as follows:

Register	Contents
0	Control program information.
1	Address of the task control block for the task that was terminated.
2-12	Unpredictable.
13	Address of a save area provided by the control program.
14	Return address (to the control program).
15	Address of the exit routine.

The exit routine is responsible for saving and restoring the registers.

,GSPV = *subpool nmbr*
,GSPL = *subpool list addr*

specifies a virtual storage subpool number less than 128 or the address of a list of virtual storage subpool numbers each less than 128. Except for subpool zero, ownership of each of the specified subpools is assigned to the new task. Although it can be specified, subpool zero cannot be transferred. When ownership of a subpool is transferred, programs of the originating task can no longer GETMAIN or FREEMAIN the associated virtual storage areas.

If GSPL is specified, the first byte of the list contains the number of remaining bytes in the list; each of the following bytes contains a virtual storage subpool number.

,SHSPV = *subpool nmbr*
,SHSPL = *subpool list addr*

specifies a virtual storage subpool number less than 128 or the address of a list of virtual storage subpool numbers each less than 128. Programs of both originating task and the new task can use the associated virtual storage areas.

If SHSPL is specified, the first byte of the list contains the number of remaining bytes in the list; each of the following bytes contains a virtual storage subpool number.

,SZERO = YES
,SZERO = NO

specifies whether subpool 0 is to be shared with the subtask. YES specifies that subpool 0 is to be shared; NO specifies that subpool 0 is not to be shared.

,TASKLIB = *dcb addr*

specifies that a task library DCB address has been supplied and is stored in TCBJLB. Otherwise, TCBJLB is propagated from the originating task. (Note: The DCB must be opened before the ATTACH macro instruction is executed.) SYS1.LINKLIB is the last library searched. If the DCB address specifies SYS1.LINKLIB, the search begins with

SYS1.LINKLIB, goes through other libraries, and ends with SYS1.LINKLIB. An 806-4 abend might occur if the requested module is in another library.

Note: DCB must reside in 24-bit addressable storage.

,STAI=(*exit addr*)
,STAI=(*exit addr,parm addr*)
,ESTAI=(*exit addr*)
,ESTAI=(*exit addr,parm addr*)
specifies whether a STAI or ESTAI SCB is to be created; any STAI/ESTAI SCBs queued to the originating task are propagated to the new task.

The *exit addr* specifies the address of the STAI or ESTAI exit routine which is to receive control if the subtask abnormally terminates; the exit routine must be in virtual storage at the time of abnormal termination. The *parm addr* is the address of a parameter list which may be used by the STAI or ESTAI exit routine.

ATTACH processing passes control to an ESTAI exit routine in the addressing mode of the issuer of the ATTACH macro instruction. Therefore, the ESTAI exit routine can execute in either 24-bit or 31-bit addressing mode. A STAI exit routine can execute only in 24-bit addressing mode. If a caller in 31-bit addressing mode specifies the STAI parameter on the ATTACH macro instruction, the caller is abended with an X'52A' completion code.

,PURGE=QUIESCE
,PURGE=NONE
,PURGE=HALT
specifies what action is to be taken with regard to I/O operations when the subtask is abnormally terminated. No action may be specified (NONE), a halting of I/O operations may be requested (HALT), or a quiescing of I/O operations may be indicated (QUIESCE).

,ASYNCH=NO
,ASYNCH=YES
specifies whether asynchronous exits are to be allowed when a subtask abnormal termination occurs.

ASYNCH=YES must be coded if:

- Any supervisor services that require asynchronous interruptions to complete their normal processing are going to be requested by the ESTAE exit routine.
- PURGE=QUIESCE is specified for any access method that requires asynchronous interruptions to complete normal input/output processing.
- PURGE=NONE is specified and the CHECK macro instruction is issued in the ESTAE exit routine for any access method that requires asynchronous interruptions to complete normal input/output processing.

Note: If ASYNCH=YES is specified and the ABEND was originally scheduled because of an error in asynchronous exit handling, an ABEND recursion will develop when an asynchronous exit handling was the cause of the failure.

,TERM=NO
,TERM=YES

specifies whether the exit routine associated with the ESTAI request is also to be scheduled in the following situations:

- CANCEL
- Forced LOGOFF
- Job step timer expiration
- Wait time limit for job step exceeded
- ABEND condition because incomplete task detached when STAE option not specified on DETACH
- ATTACH macro instruction with the ESTAI operand issued by subtask and attaching task abnormally terminates

,RELATED=*value*

specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
ATTCH1   ATTACH     EP=MYJOB,ECB=MYECB,RELATED=(DETCH1,
                    'CREATE SUBTASK')
   .
   .
   .
DETCH1   DETACH     (1),RELATED=(ATTCH1,'DETACH SUBTASK')
```

Note: The ATTACH macro instruction will fit on one line when coded, so there is no need for a continuation indicator.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion.
04	ATTACH was issued in a STAE exit; processing not completed.
08	Insufficient storage available for control block for STAI/ESTAI request; processing not completed.
0C	Invalid exit routine address or invalid parameter list address specified with STAI parameter; processing not completed.

Note: For any return code other than 00, register 1 is set to zero upon return.

Note: The program manager processing for ATTACH is performed under the new subtask, after control has been returned to the originating task. Therefore, it is possible for the originating task to obtain return code 00, and still not have the subtask successfully created (for example, if the entry name could not be found by the program manager). In such cases, the new subtask is abnormally terminated.

Example 1

Operation: Cause the program named in the list to be attached. Establish RTN as an end of task exit routine.

```
ATTACH DE=LISTNAME,ETXR=RTN
```

Example 2

Operation: Cause **PROGRAM1** to be attached, share subpool 5, wait on **WORD1** to synchronize processing with that of the subtask, and establish EXIT1 as an **ESTAI** exit.

```
ATTACH EP=PROGRAM1,SHSPV=5,ECB=WORD1,ESTAI=(EXIT1)
```

ATTACH (List Form)

Two parameter lists are used in an ATTACH macro instruction: a control program parameter list and an optional problem program parameter list. You can construct only the control program parameter list in the list form of ATTACH. Address parameters to be passed in a parameter list to the problem program can be provided using the list form of the CALL macro instruction. This parameter list can be referred to in the execute form of ATTACH.

The list form of the ATTACH macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ATTACH.
ATTACH	
b	One or more blanks must follow ATTACH.

EP=*entry name*	*entry name:* symbol.
EPLOC=*entry name addr*	*entry name addr:* A-type address.
DE=*list entry addr*	*list entry addr:* A-type address.
,DCB=*dcb addr*	*dcb addr:* A-type address.
,LPMOD=*limit prior nmbr*	*limit prior nmbr:* symbol or decimal digit.
,DPMOD=*disp prior nmbr*	*disp prior nmbr:* symbol or decimal digit.
,ECB=*ecb addr*	*ecb addr:* A-type address.
,ETXR=*exit rtn addr*	*exit rtn addr:* A-type address.
,GSPV=*subpool nmbr* ,GSPL=*subpool list addr*	*subpool nmbr:* symbol or decimal digit. *subpool list addr:* A-type address.
,SHSPV=*subpool nmbr* ,SHSPL=*subpool list addr*	*subpool nmbr:* symbol or decimal digit. *subpool list addr:* A-type address.
,SZERO=YES ,SZERO=NO	**Default:** SZERO=YES
,TASKLIB=*dcb addr*	*dcb addr:* A-type address.
,STAI=(*exit addr*) ,STAI=(*exit addr*,*parm addr*) ,ESTAI=(*exit addr*)	*exit addr:* A-type address. *parm addr:* A-type address.
,PURGE=QUIESCE ,PURGE=NONE ,PURGE=HALT	**Note:** PURGE may be specified only if STAI or ESTAI is specified. **Default** for STAI: PURGE=QUIESCE **Default** for ESTAI: PURGE=NONE
,ASYNCH=NO ,ASYNCH=YES	**Note:** ASYNCH may be specified only if STAI or ESTAI is specified. **Default** for STAI: ASYNCH=NO **Default** for ESTAI: ASYNCH=YES
,TERM=NO ,TERM=YES	**Note:** TERM may be specified only if ESTAI is specified. **Default:** TERM=NO
,RELATED=*value*	*value:* any valid macro keyword specification.
,SF=L	

The parameters are explained under the standard form of the ATTACH macro instruction, with the following exception:

,SF=L
specifies the list form of the ATTACH macro instruction.

ATTACH (Execute Form)

Two parameter lists are used in ATTACH: a control program parameter list and an optional problem program parameter list. Either or both of these parameter lists can be remote and can be referred to and modified by the execute form of ATTACH. If only the problem program parameter list is remote, parameters that require use of the control program parameter list cause that list to be constructed inline as part of the macro expansion.

The execute form of the ATTACH macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ATTACH.
ATTACH	
b	One or more blanks must follow ATTACH.
EP=*entry name* EPLOC=*entry name addr* DE=*list entry addr*	*entry name:* symbol. *entry name addr:* RX-type address, or register (2) - (12). *list entry addr:* RX-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* RX-type address, or register (2) - (12).
,LPMOD=*limit prior nmbr*	*limit prior nmbr:* symbol, decimal digit, or register (2) - (12).
,DPMOD=*disp prior nmbr*	*disp prior nmbr:* symbol, decimal digit, or register (2) - (12).
,PARAM=(*addr*) ,PARAM=(*addr*),VL=1	*addr:* RX-type address, or register (2) - (12). **Note:** *addr* is one or more addresses, separated by commas. For example, PARAM=(*addr,addr,addr*)
,ECB=*ecb addr*	*ecb addr:* RX-type address, or register (2) - (12) .
,ETXR=*exit rtn addr*	*exit rtn addr:* RX-type address, or register (2) - (12).
,GSPV=*subpool nmbr* ,GSPL=*subpool list addr*	*subpool nmbr:* symbol, decimal digit, or register (2) - (12). *subpool list addr:* RX-type address, or register (2) - (12).
,SHSPV=*subpool nmbr* ,SHSPL=*subpool list addr*	*subpool nmbr:* symbol, decimal digit, or register (2) - (12). *subpool list addr:* RX-type address, or register (2) - (12).
,SZERO=YES ,SZERO=NO	
,TASKLIB=*dcb addr*	*dcb addr:* RX-type address, or register (2) - (12).
,STAI=(*exit addr*) ,STAI=(*exit addr,parm addr*) ,ESTAI=(*exit addr*) ,ESTAI=(*exit addr,parm addr*)	*exit addr:* RX-type address, or register (2) - (12). *parm addr:* RX-type address, or register (2) - (12).
,PURGE=QUIESCE ,PURGE=NONE ,PURGE=HALT	**Note:** PURGE may be specified only if STAI or ESTAI is specified.
,ASYNCH=NO ,ASYNCH=YES	**Note:** ASYNCH may be specified only if STAI or ESTAI is specified.
,TERM=NO ,TERM=YES	**Note:** TERM may be specified only if ESTAI is specified.
,RELATED=*value*	*value:* any valid macro keyword specification.
,MF=(E,*prob addr*) ,SF=(E,*ctrl addr*) ,MF=(E,*prob addr*),SF=(E,*ctrl addr*)	*prob addr:* RX-type address, or register (1) or (2) - (12). *ctrl addr:* RX-type address, or register (2) - (12) or (15).

The parameters are explained under the standard form of the ATTACH macro instruction, with the following exceptions:

,MF = (E,*prob addr***)**
,SF = (E,*ctrl addr***)**
,MF = (E,*prob addr***),SF = (E,***ctrl addr***)**
specifies the execute form of the ATTACH macro instruction using either a remote problem program parameter list or a remote control program parameter list. Any problem program or control program parameters are provided in parameter lists expanded inline.

Notes:

1. *If STAI is specified on the execute form, the following fields are overlaid in the control program parameter list: exit addr, parm addr, PURGE, and ASYNCH. If parm addr is not specified, zero is used; if PURGE or ASYNCH are not specified, defaults are used.*

2. *If ESTAI is specified on the execute form, then the following fields are overlaid: exit addr, parm addr, PURGE, ASYNCH, and TERM. If parm addr is not specified, zero is used; if PURGE, ASYNCH, or TERM are not specified, defaults are used.*

3. *If the STAI or ESTAI is to be specified, it must be completely specified on either the list or execute form, but not on both forms.*

4. *If SZERO is not specified on the list or execute form, the default is SZERO=YES. If SZERO=NO is specified on either the list form or a previous execute form using the same SF=list, then SZERO=YES is ignored for any following execute forms of the macro. Once SZERO=NO is specified, it is in effect for all users of that list.*

CALL — Pass Control to a Control Section

The CALL macro instruction passes control to a control section at a specified entry point as follows:

- OVERLAY: The overlay segment containing the designated entry point is brought into virtual storage if required, and control is passed to the segment.

  Refer to *Linkage Editor and Loader* for details on overlay. Do not use the CALL macro instruction in an asynchronous exit routine.

- NON-OVERLAY: If a symbol is designated, the linkage editor includes the load module containing that entry point in the same load module containing the CALL instruction. When the CALL macro instruction is executed, control is passed to the control section at the specified entry point.

The linkage relationship established when control is passed is the same as that created by a BAL instruction; that is, the issuing program expects control to be returned. The control program is not involved in passing control, so the reusability of the called program must be maintained by the user.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction. You cannot use the CALL macro instruction to pass control to a program in a different addressing mode.

An address parameter list can be constructed and a calling sequence identifier can be provided.

The standard form of the CALL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CALL.
CALL	
b	One or more blanks must follow CALL.

entry name	*entry name:* symbol or register (15).
,(*addr*) ,(*addr*),VL	*addr:* A-type address, or register (2) - (12). **Note:** *addr* is one or more addresses, separated by commas. For example, (*addr*,*addr*,*addr*)
,ID = *id nmbr*	*id nmbr:* symbol or decimal digit, with a maximum value of 4095.

The parameters are explained as follows:

entry name
specifies the entry name to be given control.

,(*addr*)
,(*addr*),VL
specifies address(es) to be passed to the control program. Each address is expanded inline to a fullword on a fullword boundary, in the order designated. Register 1 contains the address of the first parameter when the program is given control. (If this parameter is not coded, register 1 is not altered.)

VL should be coded only if the called program can be passed a variable number of parameters. VL causes the high-order bit of the last address parameter to be set to 1; the bit can be checked to find the end of the list.

,ID=*id nmbr*
specifies an identifier useful for debugging purposes only. The last fullword of the macro expansion is a NOP instruction containing the identifier value in bytes 3 and 4.

Upon entry to the called program, the register contents are as follows:

Register	Meaning
1	Address of parameter list, if present.
14	Return address.
15	Entry address of called program.

Example 1

Operation: Call the entry point contained in register 15, and pass three addresses to the control program.

```
CALL      (15),(ADDR1,ADDR2,ADDR3)
```

CALL (List Form)

The list form of the CALL macro instruction is used to construct a nonexecutable problem program parameter list. This list form generates only ADCONs of the address parameters. This problem program parameter list can be referred to in the execute form of a CALL, LINK, ATTACH, or XCTL macro instruction.

The list form of the CALL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CALL.
CALL	
b	One or more blanks must follow CALL.
,(*addr*)	*addr:* A-type address.
,(*addr*),VL	**Note:** *addr* is one or more addresses, separated by commas. For example, (*addr*,*addr*,*addr*)
,MF=L	

The parameters are explained under the standard form of the CALL macro instruction, with the following exception:

,MF=L
specifies the list form of the CALL macro instruction.

CALL (Execute Form)

A remote problem program parameter list is referred to and can be modified by the execute form of the CALL macro instruction. Only executable instructions and a VCON of the entry point are generated.

The execute form of the CALL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CALL.
CALL	
b	One or more blanks must follow CALL.

entry name	*entry name:* symbol or register (15).
,(*addr*) ,(*addr*),VL	*addr:* RX-type address, or register (2) - (12). **Note:** *addr* is one or more addresses, separated by commas. For example, (*addr,addr,addr*)
,ID = *id nmbr*	*id nmbr:* symbol or decimal digit, with a maximum value of 4095.
,MF = (E,*prob addr*)	*prob addr:* RX-type address, or register (1) or (2) - (12).

The parameters are explained under the standard form of the CALL macro instruction, with the following exception:

,MF = (E,*prob addr*)
specifies the execute form of the CALL macro instruction. This form uses a remote problem program parameter list. If the address parameters are also specified in this form, the ADCONS of the parameter are placed on contiguous fullword boundaries beginning at the address specified in the MF parameter, and sequentially overlaying corresponding fullwords in the existing list.

CHAP — Change Dispatching Priority

CHAP changes the dispatching priority of the task or any of its subtasks relative to the other tasks in the address space. It does not change the priority relative to other tasks in the system. CHAP may also change the limit priority of a subtask. (See the section "Priorities" in this publication.) The algebraic sum of the priority value and the dispatching priority of the subject task determines the new dispatching priority.

- If the subject task is the task executing CHAP, its dispatching priority is set equal to the sum of the priority value and the dispatching priority. This value is not set at less than zero or greater than the limit priority for the task. Its limit priority is unaffected.

- If the subject task is a subtask of the task executing CHAP, its dispatching priority is set equal to the sum of the priority value and the dispatching priority. This value is not set at less than zero or greater than the limit priority of the task executing CHAP. After this modification, if the subtask's dispatching priority exceeds its limit priority, the limit priority is made equal to the dispatching priority.

The CHAP macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CHAP.
CHAP	
b	One or more blanks must follow CHAP.

priority value	*priority value:* symbol, decimal digit, or register (0) or (2) - (12).
,*'S'* ,*tcb addr*	*tcb addr:* RX-type address, or register (1) or (2) - (12). **Default:** 'S'
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

priority value
specifies the signed value to be added to the dispatching priority of the specified task. If the value is negative and contained in a register, it must be in two's complement form.

,'S'
,tcb addr
specifies the address of a fullword on a fullword boundary containing the address of a task control block (TCB) for a subtask of the active task. If 'S' is coded or assumed, the dispatching priority of the active task is updated.

Note: TCB must reside in 24-bit addressable storage.

,RELATED=*value*
specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
CHAPUP      CHAP 1,'S',RELATED=(CHAPDOWN,'UP PRIORITY')
  .
  .
  .
CHAPDOWN    CHAP -1,'S',RELATED=(CHAPUP,
            'RESUME INITIAL PRIORITY')
```

Note: The second CHAP macro instruction will fit on one line when coded, so there is no need for a continuation indicator.

Example 1

Operation: Lower by 2 the dispatching priority of the subtask TCB, whose address is in a fullword which is addressed by register 1. The subtask TCB will be repositioned on the dispatching queue in accordance with its new dispatching priority.

```
CHAP        -2,(1)
```

Example 2

Operation: Cause the TCB of the task issuing CHAP to be repositioned at the bottom of the group of TCBs on the dispatching queue for the address space, having the same dispatching priority as that task.

```
CHAP        0
```

CPOOL — Perform Cell Pool Services

The CPOOL macro instruction creates a cell pool, obtains or returns a cell to the cell pool, or deletes the previously built cell pool, according to the function requested. Problem state, non-system key users cannot create cell pools in subpools greater than 127. On entry to the CPOOL macro instruction, users who specify the parameters: BUILD, DELETE, or REGS=SAVE must pass the address of a 72-byte save area in register 13.

The CPOOL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CPOOL.
CPOOL	
b	One or more blanks must follow CPOOL.

BUILD GET FREE DELETE	
,UNCOND ,U ,COND ,C	**Default:** UNCOND **Note:** This parameter can be specified only with the GET keyword.
,PCELLCT=*primary cell count*	*cell count:* symbol, decimal digit, or register (0), (2) - (12). **Note:** This parameter can be specified only with the BUILD keyword.
,SCELLCT=*secondary cell count*	**Default:** PCELLCT **Note:** This parameter can be specified only with the BUILD keyword.
,CSIZE=*cell size*	*cell size:* symbol, decimal digit, or register (0), (2) - (12). **Note:** This parameter can be specified only with the BUILD keyword.
,SP=*subpool number*	*subpool number:* symbol, decimal digit 0-127, or register (0), (2) - (12). **Note:** This parameter can be specified only with the BUILD keyword. **Default:** SP=0
,LOC=BELOW ,LOC=(BELOW,ANY) ,LOC=ANY ,LOC=(ANY,ANY) ,LOC=RES ,LOC=(RES,ANY)	**Default:** LOC=RES **Note:** This parameter can be specified only with the BUILD keyword.
,CPID=*pool id*	*pool id:* RX-type address or register (0), (2) - (12). **Note:** This parameter must be specified with the GET, FREE, and DELETE keywords but is optional with the BUILD keyword.
,CELL=*cell addr*	*cell addr:* RX-type address or register (2) - (12). **Note:** This parameter is required with the FREE keyword, is optional with the GET keyword, and cannot be specified with the BUILD and DELETE keywords.
,HDR=*hdr*	*hdr:* character string enclosed in single quotes, RX-type address, or register (0), (2) - (12). **Default:** 'CPOOL CELL POOL' **Note:** This parameter can be specified only with the BUILD keyword.
,REGS=SAVE ,REGS=USE	**Default:** REGS=SAVE **Note:** This parameter can be specified only with the GET or FREE keywords.

The parameters are explained as follows:

BUILD
GET
FREE
DELETE
specifies the cell pool service to be performed.

BUILD creates a cell pool in a specified subpool by allocating storage and chaining the cells together. It returns an identifier (CPID) to be used with GET, FREE, and DELETE requests. Therefore, BUILD must be done before GET, FREE, or DELETE.

GET attempts to obtain a cell from the previously built cell pool. This request can be conditional or unconditional as described under the UNCOND/COND keyword.

FREE returns a cell to the cell pool.

DELETE deletes a previously built cell pool and frees storage for the initial extent, all secondary extents, and all pool control blocks.

,UNCOND
,U
,COND
,C
when used with GET specifies whether the request for a cell is conditional or unconditional. If COND or C is specified and the cell pool is empty, the CPOOL service routine returns to the caller without a cell and places a zero in the return field of the cell address. If UNCOND or U is specified and the cell pool is empty, the CPOOL service routine extends the pool in order to obtain a cell for the caller.

,PCELLCT = *primary cell count*
specifies the number of cells expected to be needed in the initial extent of the cell pool. The CPOOL service module uses PCELLCT and cell size (CSIZE) to determine the optimum number of cells to provide in order to make effective use of virtual and real storage.

,SCELLCT = *secondary cell count*
specifies the number of cells expected to be in each secondary or non-initial extent of the cell pool. The CPOOL service routine uses SCELLCT and CSIZE to determine the optimum number of cells to provide in order to make effective use of virtual and real storage.

,CSIZE = *cell size*
specifies the number of bytes in each cell of the cell pool. If CSIZE is a multiple of 8, the cell resides on doubleword boundaries. If CSIZE is a multiple of 4, the cell resides on word boundaries. The minimum value of CSIZE is 4 bytes.

,SP = *subpool number*
specifies the subpool from which the cell pool is to be obtained. If a register or variable is specified, the subpool number is taken from bits 24-31.

,LOC=BELOW
,LOC=(BELOW,ANY)
,LOC=ANY
,LOC=(ANY,ANY)
,LOC=RES
,LOC=(RES,ANY)

specifies the location of virtual storage and real storage for the cell pool.

Note: The location of real storage using this parameter is only guaranteed after the storage is fixed.

LOC=BELOW indicates that virtual and real storage are to be allocated below 16 Mb.

LOC=(BELOW,ANY) indicates that virtual storage is to be allocated below 16 Mb and real storage can be anywhere.

LOC=ANY and LOC=(ANY,ANY) indicate that both virtual and real storage can be located anywhere.

LOC=RES indicates that the location of virtual and real storage depends on the location of the issuer of the macro. If the issuer resides below 16 Mb, virtual and real storage are allocated below 16 Mb; if the issuer resides above 16 Mb, virtual and real storage can be located anywhere.

LOC=(RES,ANY) indicates that the location of virtual storage depends on the location of the issuer of the macro. If the issuer resides below 16 Mb, virtual storage is allocated below 16 Mb; if the issuer resides above 16 Mb, virtual storage is allocated anywhere. Real storage can be located anywhere.

Note: Callers executing in 24-bit addressing mode could perform BUILD request services for cell pools located in storage above 16 Mb by specifying LOC=ANY or LOC=(ANY,ANY).

,CPID=*pool id*

specifies the address or register containing the cell pool identifier that is returned to the caller after the pool is created using CPOOL BUILD. The issuer must specify CPID on all subsequent CPOOL requests containing the keywords GET, FREE, or DELETE.

,CELL=*cell addr*

specifies the address or register where the cell address is returned to the caller on a GET or FREE request.

,HDR=*hdr*

specifies a 24-byte header, which is placed in the header of each initial and secondary extent. The header can contain user-supplied information that would be useful in a dump.

,REGS=SAVE
,REGS=USE

indicates whether or not registers 2-12 are to be saved. If REGS=SAVE is specified, the registers are saved in a 72-byte user-supplied save area pointed to by register 13. If REGS=USE is specified, the registers are not saved.

The contents of the registers on return from this macro depends on the parameters specified.

Register(s)	Comment
0	Contains the cell pool identification (CPID)
1	Contains the address of the cell that was obtained if GET was specified; contains zero if GET conditional was specified and the cell could not be obtained
2-12	Saved for BUILD and DELETE requests or if REGS=SAVE is specified
5-13	Saved if GET conditional or FREE is specified with REGS=USE
13	Saved if GET unconditional and REGS=USE, BUILD, or DELETE is specified

Example 1

Operation: Create a cell pool containing 40-byte cells from subpool 2. Allow for 10 cells in the initial extent and 20 cells in all subsequent extents of the cell pool.

```
CPOOL BUILD,PCELLCT=10,SCELLCT=20,CSIZE=40,SP=2
```

Example 2

Operation: Unconditionally obtain a cell pool, specifying the pool ID in register 2. Do not save the registers.

```
CPOOL GET,U,CPID=(2),REGS=USE
```

Example 3

Operation: Free a cell specifying the pool ID in register 2 and the cell address in register 3.

```
CPOOL FREE,CPID=(2),CELL=(3)
```

Example 4

Operation: Delete a cell pool, specifying the pool ID in register 2.

```
CPOOL DELETE,CPID=(2)
```

CPOOL — (List Form)

The list form of the CPOOL macro instruction builds a non-executable parameter list that can be referred to by the execute form of the CPOOL macro.

The list form of the CPOOL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CPOOL.
CPOOL	
b	One or more blanks must follow CPOOL.
BUILD	
,PCELLCT=*primary cell count*	*cell count:* symbol, decimal digit, or register (0), (2) - (12). **Note:** PCELLCT must be specified on either the list or the execute form of the macro.
,SCELLCT=*secondary cell count*	**Default:** PCELLCT
,CSIZE=*cell size*	*cell size:* symbol, decimal digit, or register (0), (2) - (12). **Note:** CSIZE must be specified on either the list or the execute form of the macro.
,SP=*subpool number*	*subpool number:* symbol, decimal digit 0-127, or register (0), (2) - (12). **Default:** SP=0
,LOC=BELOW ,LOC=(BELOW,ANY) ,LOC=ANY ,LOC=(ANY,ANY) ,LOC=RES ,LOC=(RES,ANY)	**Default:** LOC=RES
,CPID=*pool id*	*pool id:* A-type address or register (0), (2) - (12).
,HDR=*hdr*	*hdr:* character string enclosed in single quotes, A-type address, or register (0), (2) - (12).
,MF=L	

The parameters are explained under the standard form of the CPOOL macro instruction with the following exception:

,MF=L
specifies the list form of the CPOOL instruction.

CPOOL — (Execute Form)

The execute form of the CPOOL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CPOOL.
CPOOL	
b	One or more blanks must follow CPOOL.
BUILD	
,PCELLCT=*primary cell count*	*cell count:* symbol, decimal digit, or register (0), (2) - (12). **Note:** PCELLCT must be specified on either the list or the execute form of the macro.
,SCELLCT=*secondary cell count*	**Default:** PCELLCT
,CSIZE=*cell size*	*cell size:* symbol, decimal digit, or register (0), (2) - (12). **Note:** CSIZE must be specified on either the list or the execute form of the macro.
,SP=*subpool number*	*subpool number:* symbol, decimal digit 0-127, or register (0), (2) - (12). **Default:** SP=0
,LOC=BELOW ,LOC=(BELOW,ANY) ,LOC=ANY ,LOC=(ANY,ANY) ,LOC=RES ,LOC=(RES,ANY)	**Default:** LOC=RES
,CPID=*pool id*	*pool id:* RX-type address or register (0), (2) - (12).
,HDR=*hdr*	*hdr:* character string enclosed in single quotes, RX-type address, or register (0), (2) - (12).
,MF=(E,*ctrl prog*)	*ctrl prog:* RX-type address or register (0) - (12).

The parameters are explained under the standard form of the CPOOL macro instruction with the following exception:

,MF=(E,*ctrl prog*)
specifies the execute form of the CPOOL instruction.

CPUTIMER — Provide Current CPU Timer Value

The CPUTIMER macro instruction provides the current CPU timer value for this processor. This value consists of the time remaining in a time interval established by the STIMER macro instruction. If there is no outstanding time interval, the value returned by the macro instruction is meaningless.

The caller of the CPUTIMER macro instruction must provide the address of a 72-byte save area in register 13.

The CPUTIMER macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede CPUTIMER.
CPUTIMER	
b	One or more blanks must follow CPUTIMER.
TU,*stor addr*	**Default:** TU
MIC,*stor addr*	*stor addr:* RX-type address, or register (1), (2) - (12).
,ERRET=*err rtn addr*	*err rtn addr:* RX-type address, or register (2) - (12).

The parameters are explained as follows:

TU,*stor addr*
MIC,*stor addr*
specifies the form in which the remaining time interval is to be returned to the caller. This value is returned as an unsigned 64-bit binary number, at the address specified by *stor addr*. *stor addr* must be the start of a double word area on a double word boundary and it must be a 31-bit address.

If TU is specified, the timer value is returned to the caller in timer units. The low-order bit of the timer value is approximately equal to 26.04166 microseconds (one timer unit).

If MIC is specified, the timer value is returned to the caller in microseconds. Bit 51 of the timer value is equivalent to 1 microsecond.

The resolution of CPU timer is model dependent. See *Principles of Operation* for a description of the CPU timer.

,ERRET=*err rtn addr*
specifies the 31-bit address of the routine to be given control when the CPUTIMER function cannot be performed. If this parameter is omitted, the CPUTIMER function returns a code in general register 15 indicating why the function could not be performed. The error routine executes in the addressing mode of the issuer of the CPUTIMER macro instruction.

The return codes are as follows:

Hexadecimal Code	Meaning
0	The function was performed.
4	The function was not performed because the user-specified area was not on a double word boundary.
8	The function was not performed because the user supplied an invalid address.
0C	The function was not performed because the value of the CPU timer was not usable.
10	The function was not performed because a machine check occurred.
14	The function was not performed because a program check occurred.

Example 1

Operation: Place the value of the CPU timer in microseconds in location TIMELEFT.

```
CPUTIMER MIC,TIMELEFT
```

Example 2

Operation: Store the value of the CPU timer in time units in the location addressed by register 1.

```
CPUTIMER TU,(1)
```

Example 3

Operation: Store the value of the CPU timer in timer units in location TIMELEFT. If an error occurs, transfer control to the error routine labeled ERREXIT.

```
CPUTIMER ,TIMELEFT,ERRET=ERREXIT
```

Example 4

Operation: Place the value of the CPU timer in microseconds in the location addressed by register 1. If an error occurs, transfer control to the address in register 2.

```
CPUTIMER MIC,(1),ERRET=(2)
```

DELETE — Relinquish Control of a Load Module

The DELETE macro instruction cancels the effect of a previous LOAD macro instruction. If DELETE cancels the only outstanding LOAD request for the module and no other requirements exist for the module, the virtual storage occupied by the load module is released and is available for reassignment by the control program.

The entry name specified in the DELETE macro instruction must be the same as that specified in the LOAD macro instruction that brought the load module into storage. Also, the DELETE macro instruction must be issued by the same task that issued the LOAD macro instruction.

Any module loaded by a task will not be removed from virtual storage until the DELETE macro instruction is issued or end of task is reached. In addition, any module loaded by a system task will not be removed from virtual storage until a DELETE macro instruction is issued by a system task or end of task is reached.

The DELETE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DELETE.
DELETE	
b	One or more blanks must follow DELETE.

EP=*entry name*	*entry name:* symbol.
EPLOC=*entry name addr*	*entry name addr:* RX-type address, or register (0) or (2) - (12).
DE=*list entry addr*	*list entry addr:* RX-type address, or register (0) or (2) - (12).
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

EP=*entry name*
EPLOC=*entry name addr*
DE=*list entry addr*

specifies the entry name, the address of the entry name, or the address of a 60-byte list entry for the entry name that was constructed using the BLDL macro instruction. If EPLOC is coded, the name must be padded to eight bytes, if necessary.

,RELATED=*value*

specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
LOAD1    LOAD       EP=APGIOHK1,RELATED=(DEL1,
                    'LOAD APGIOHK1')
   .
   .
   .
DEL1     DELETE     EP=APGIOHK1,RELATED=(LOAD1,
                    'DELETE APGIOHK1')
```

Note: Each of these macro instructions will fit on one line when coded, so there is no need for a continuation indicator.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion of requested function.
04	Request was not issued for this module, or attempt was made to delete a system module.

Example 1

Operation: Remove a module (PGMTOVLY) from virtual storage.

```
DELETE EP=PGMTOVLY
```

DEQ — Release a Serially Reusable Resource

The DEQ macro instruction removes control of one or more serially reusable resources from the active task. Register 15 is set to 0 if the request is satisfied. An unconditional request to release a resource from a task that is not in control of the resource, or a request that contains invalid parameters results in abnormal termination of the task.

Note: When global resource serialization is active, the SYSTEM inclusion resource name list and the SYSTEMS exclusion resource name list are searched for every resource specified with a scope of SYSTEM or SYSTEMS. A resource whose name appears in one of these resource name lists might have its scope changed from the scope that appears on the macro instruction. See *Planning: Global Resource Serialization* for more information.

The standard form of the DEQ macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DEQ.
DEQ	
b	One or more blanks must follow DEQ.
(	
qname addr	*qname addr:* A-type address, or register (2) - (12).
,*rname addr*	*rname addr:* A-type address, or register (2) - 12).
, ,*rname length*	*rname length:* symbol, decimal digit, or register (2) - (12). **Note:** *rname length* must be coded if a register is specified for *rname addr*.
, ,STEP ,SYSTEM ,SYSTEMS	**Default:** STEP
)	
,RET=HAVE ,RET=NONE	**Default:** RET=NONE
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

(
: specifies the beginning of the resource(s) description.

qname addr
: specifies the address in virtual storage of an 8-character name. The name can contain any valid hexadecimal digits. The *qname* must be the same name specified for the resource in an ENQ macro instruction.

,rname addr
: specifies the address in virtual storage of the name used in conjunction with *qname* and scope to represent the resource acquired by a previous ENQ macro instruction. The name can be qualified, must be from 1 to 255 bytes long, and can contain any valid hexadecimal digits. The *rname* must be the same name specified for the resource in an ENQ macro instruction.

,rname length
specifies the length of the *rname* described above. The length must have the same value as specified in the previous ENQ macro instruction. If this parameter is omitted, the assembled length of the *rname* is used. You can specify a value between 1 and 255 to override the assembled length, or you may specify a value of 0. If 0 is specified, the length of the *rname* must be contained in the first byte at the *rname addr* specified above.

,
,STEP
,SYSTEM
,SYSTEMS
specifies the scope of the resource. You must specify the same STEP, SYSTEM, or SYSTEMS option as you used in the ENQ macro instruction requesting the resource.

)
specifies the end of the resource(s) description.

Note: The parameters *qname addr*, *rname addr*, *rname length*, and the scope can be repeated within a single set of parentheses to indicate multiple resources. These parameters can be repeated until there is a maximum of 255 characters including the parentheses.

,RET=HAVE
,RET=NONE
specifies that the request for releasing the resources named in DEQ is to be conditional (HAVE) or unconditional (NONE). If this parameter is omitted, the request for release is unconditional, and the active task is abnormally terminated if it has not been assigned control of the resources.

HAVE specifies that the request to release the resources named in the DEQ macro instruction is to be honored only if the active task has been assigned control of the resources. A return code is set if the resource is not held.

NONE specifies an unconditional request to release the resources. The active task is abnormally terminated if it has not been assigned control of the resources. If the parameter is omitted, NONE is the default.

,RELATED=*value*
specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and can be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
ENQUEUE  ENQ    (MAJOR,MINOR,S,8,STEP),                    X
                RELATED=(DEQUEUE,'OBTAIN RESOURCE')
   .
   .
   .
DEQUEUE  DEQ    (MAJOR,MINOR,8,STEP),                      X
                RELATED=(ENQUEUE,'RELEASE RESOURCE')
```

Return codes are provided by the control program only if RET=HAVE is designated. If all of the return codes for the resources named in DEQ are 0, register 15 contains 0. If any of the return codes are not 0, register 15 contains the address of a virtual storage area containing the return codes as shown in Figure 50. The return codes are placed in the parameter list resulting from the macro expansion in the same sequence as the resource names in the DEQ macro instruction. The return codes are shown in Figure 51.

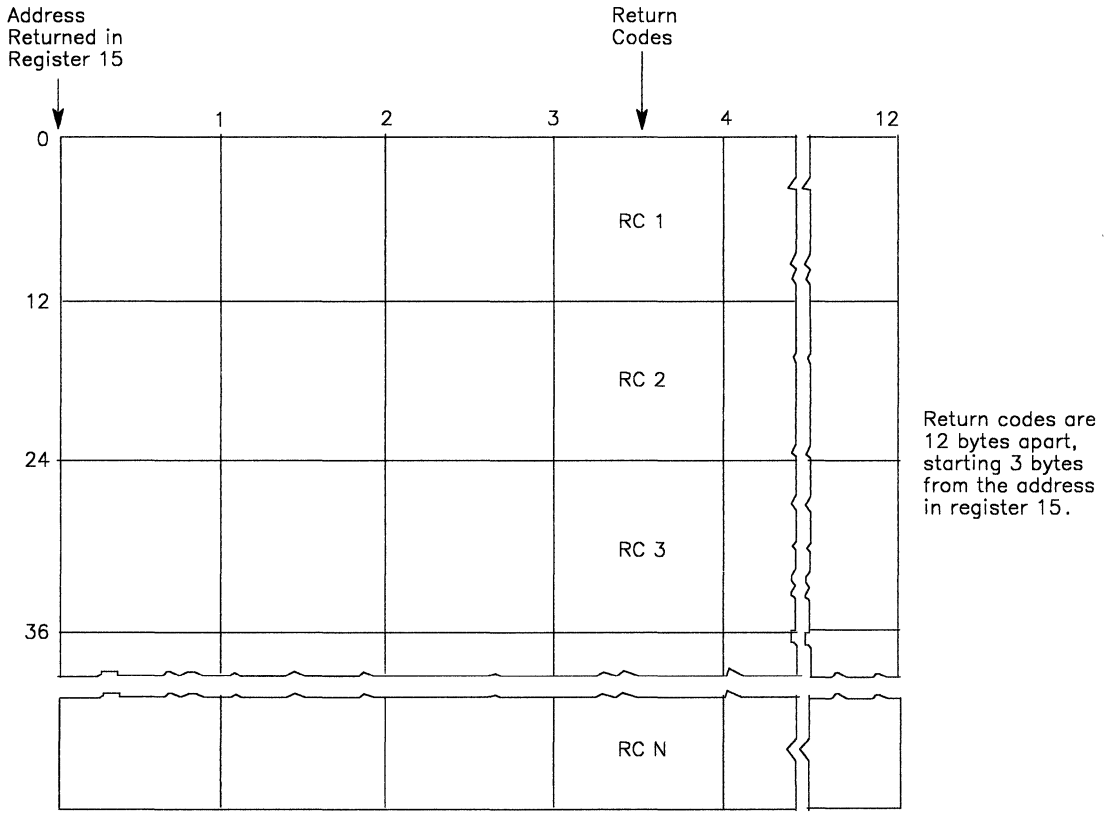

Figure 50. Return Code Area Used by DEQ

Hexadecimal Code	Meaning
0	The resource has been released.
4	The resource has been requested for the task, but the task has not been assigned control. The task is not removed from the wait condition. (This return code could result if DEQ is issued within an exit routine which was given control because of an interruption.)
8	Control of the resource has not been requested by the active task, or the resource has already been released.

Figure 51. DEQ Macro Instruction Return Codes

Example 1

Operation: Release control of the resource in Example 1 of ENQ, (later in this section) if it has been assigned to the current task. The length of the rname is explicitly defined as 9 characters.

```
DEQ (MAJOR1,MINOR1,9,STEP),RET=HAVE
```

Example 2

Operation: Unconditionally release control of the resources in Example 2 of ENQ. The length of the *rname* for the first resource is 3 characters and the length of the *rname* for the third resource is 8 characters. Allow the length of the second resource to default to its assembled length.

```
DEQ (MAJOR4,MINOR4,3,STEP,MAJOR2,MINOR2,,SYSTEM,          X
     MAJOR3,MINOR3,8,SYSTEMS)
```

DEQ (List Form)

Use the list form of the DEQ macro instruction to construct a DEQ parameter list. The number of *qname*, *rname*, and scope combinations in the list form of DEQ must be equal to the maximum number of *qname*, *rname* and scope combinations in any execute form of DEQ that refers to that list form.

The list form of the DEQ macro instruction is written as follows:

```
    name                        name: symbol. Begin name in column 1.
b                               One or more blanks must precede DEQ.
DEQ
b                               One or more blanks must follow DEQ.

(
    qname addr                  qname addr: A-type address.
    ,                           rname addr: A-type address.
    ,rname addr
    ,                           rname length: symbol or decimal digit.
    ,rname length

    ,
    ,STEP                       Default: STEP
    ,SYSTEM
    ,SYSTEMS

)
    ,RET=HAVE
    ,RET=NONE
    ,RELATED=value              value: any valid macro keyword specification.
,MF=L
```

The parameters are explained under the standard form of the DEQ macro instruction, with the following exception:

,MF=L
specifies the list form of the DEQ macro instruction.

DEQ (Execute Form)

A remote control program parameter list is used in, and can be modified by, the execute form of the DEQ macro. The parameter list can be generated by the list form of either the DEQ or the ENQ macro instruction. a1.The execute form of the DEQ macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DEQ.
DEQ	
b	One or more blanks must follow DEQ.
(	**Note:** (and) are the beginning and end of a parameter list. The entire list is optional. If nothing in the list is desired, the (,), and all parameters between (and) should not be specified. If something in the list is desired, the (,), and all parameters in the list should be specified as indicated at the left.
qname addr	*qname addr:* RX-type address, or register (2) - (12).
, ,*rname addr*	*rname addr:* RX-type address, or register (2) - (12).
, ,*rname length*	*rname length:* symbol, decimal digit or register (2) - (12).
, ,STEP ,SYSTEM ,SYSTEMS	**Default:** STEP
)	**Note:** See note opposite (above.
,RET=HAVE ,RET=NONE	**Default:** RET=NONE
,RELATED=*value*	*value:* any valid macro keyword specification.
,MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) - (12).

The parameters are explained under the standard form of the DEQ macro instruction, with the following exception:

,MF=(E,*ctrl addr*)
specifies the execute form of the DEQ macro instruction using a DEQ parameter list.

DETACH — Detach a Subtask

The DETACH macro instruction removes from the system a subtask created by an ATTACH macro instruction that specified the ECB or ETXR parameter. Each subtask created in this manner must be removed from the system before the originating task terminates. Failure to remove these subtasks causes abnormal termination of the originating task and all of its subtasks. Issuing a DETACH macro instruction that specifies a subtask created without the ECB or ETXR parameter also causes abnormal termination of the originating task when the specified subtask has already terminated. Issuing a DETACH macro instruction that specifies a subtask that has not terminated causes termination of that subtask and all of its subtasks. A DETACH macro instruction can be issued only for subtasks created by the active task.

The DETACH macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DETACH.
DETACH	
b	One or more blanks must follow DETACH.
tcb addr	*tcb addr:* symbol, RX-type address, or register (1) or (2) - (12).
,STAE=NO ,STAE=YES	**Default:** STAE=NO
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

tcb addr
specifies the address of a fullword on a fullword boundary containing the address of the task control block for the subtask to be removed from the system.

Note: *tcb addr* specifies a storage location below 16 Mb.

,STAE=NO
,STAE=YES
specifies whether the exit routine specified in a STAE macro instruction issued by the subtask, or STAI/ESTAE/ESTAI exits existing for the subtasks, is or is not to be given control if the subtask is detached before it has been terminated. If a retry routine is specified by the STAE exit routine, it is not given control.

,RELATED=*value*
specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
ATTCH1   ATTACH  EP=MYJOB,ECB=MYECB,RELATED=(DETCH1,
                 'CREATE SUBTASK')
         ST      1,TCBADDR           SAVE TCB ADDRESS
         WAIT    1,MYECB             WAIT FOR SUBTASK
                                     TO COMPLETE
DETCH1   DETACH  TCBADDR,RELATED=(ATTCH1,'DETACH SUBTASK')
```

Note: The ATTACH macro instruction will fit on one line when coded, so there is no need for a continuation indicator.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion.
04	An incomplete subtask was detached with STAE=YES specified; DETACH processing successfully completed.

Example 1

Operation: Cause the subtask to be removed from the address space. The address of the TCB is in the fullword labeled SAVEWORD.

```
DETACH SAVEWORD
```

Example 2

Operation: In addition to causing the subtask to be removed from the address space, give control to the most recent STAE exit established by the subtask if the subtask has not yet been terminated.

```
DETACH (1),STAE=YES
```

DIV — Data-in-Virtual

The DIV macro lets you access a data object on permanent storage via paging I/O and process this object through normal virtual storage addressing. Data-in-virtual maps the object onto a single virtual address range so your program can view it as beginning at a virtual location and occupying a consecutive virtual address range. The DIV macro performs the following eight services:

- IDENTIFY
- ACCESS
- MAP
- RESET
- SAVE
- UNMAP
- UNACCESS
- UNIDENTIFY

The standard form of the DIV macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DIV.
DIV	
b	One or more blanks must follow DIV.

	Valid parameters:
IDENTIFY	ID, TYPE, DDNAME
ACCESS	ID, MODE, SIZE, LOCVIEW
MAP	ID, AREA, OFFSET, SPAN, RETAIN
RESET	ID, OFFSET, SPAN
SAVE	ID, OFFSET, SPAN, SIZE
UNMAP	ID, AREA, RETAIN
UNACCESS	ID
UNIDENTIFY	ID
,ID = *addr*	
,AREA = *addr*	*addr:* RX type address or register (2) - (12)
,DDNAME = *addr*	
,LOCVIEW = MAP ,LOCVIEW = NONE	**Default:** LOCVIEW = NONE
,MODE = READ ,MODE = UPDATE	
,OFFSET = *addr* ,OFFSET = *	**Default:** OFFSET = 0
,RETAIN = YES ,RETAIN = NO	**Default:** RETAIN = NO
,SIZE = *addr* ,SIZE = *	See explanation of parameters if omitted
,SPAN = *addr* ,SPAN = * ,TYPE = DA	See explanation of parameters if omitted

The IDENTIFY, ACCESS, MAP, SAVE, RESET, UNMAP, UNACCESS and UNIDENTIFY parameters, which designate the services of the DIV macro, are mutually exclusive. You can select only one. The parameters are explained as follows:

IDENTIFY
selects the data-in-virtual object (linear data set) that you want to process. When you specify IDENTIFY, you must also specify ID, DDNAME and TYPE. ID specifies the address of an eight-byte field, DDNAME specifies the object, and TYPE specifies the storage format of the DDNAME. The IDENTIFY service creates a unique eight-byte internal name that is returned at the address specified by the ID parameter. This name is a token that represents the use of the selected object within your task. When you invoke other data-in-virtual services, you use this token as the ID input parameter.

ACCESS
requests permission to access a data-in-virtual object. When you specify ACCESS, you must also specify ID and MODE and you may optionally specify SIZE. The ID parameter, which provides the address of the unique name that was returned by the IDENTIFY service, indicates the object you want to access. If SIZE is specified, the macro returns the current size of the object in the location that SIZE designates. If specified, MODE indicates whether the object will be accessed for reading or updating. LOCVIEW=MAP specifies that the map is to be protected from data changes in the object. LOCVIEW=NONE specifies that the map is not to be protected from data changes in the object.

MAP
specifies a request to establish addressability to the object in a specified range of virtual storage, called the virtual window. When you specify MAP, you must also specify ID and AREA, and you may optionally specify OFFSET, SPAN, and RETAIN. The ID parameter, which provides the address of the unique name that was returned by the IDENTIFY service, selects the object that you want to map. AREA indicates the starting address of the virtual window. OFFSET and SPAN specify the range of blocks on the object that is to be mapped to the window. RETAIN indicates whether or not data previously existing in the virtual window is kept, or replaced by data from the object data.

RESET
releases changes made in the window since the last SAVE operation. When you specify RESET, you must also specify ID, and you may optionally specify OFFSET and SPAN. ID, which provides the address of the unique name that was returned by the IDENTIFY service, indicates the virtual window that you want to reset. OFFSET and SPAN specify the range of blocks on the object that corresponds to the virtual window. Data in pages of mapped virtual windows that correspond to the RESET range will be replaced. If the window corresponds to blocks on the object, the data is replaced by the current contents of the object. If the window corresponds to blocks offset beyond the end of the object, the data is not relevant (as if the pages had just been obtained by a GETMAIN). No change to the object itself will be made by a RESET.

SAVE
specifies that data from the window is to be saved. Saving is accomplished by writing changed pages from the window to the corresponding blocks of the object. When you specify SAVE, you must also specify ID, and you may optionally specify OFFSET, SPAN and SIZE. ID, which specifies the unique name that was returned by the IDENTIFY service, selects the object into which the blocks are written. OFFSET and SPAN specify the data blocks, relative to the beginning of the object, that are to be saved. Changed pages from the window are then written into those blocks.

UNMAP
specifies a request to terminate a virtual window by removing the correspondence between virtual pages in the window and blocks on the object. After the UNMAP is complete, the contents of the pages depend on the value you specify for RETAIN; the virtual pages in the former window either retain a copy of the data that was on the corresponding blocks of the object, or appear as if they had just been obtained by a GETMAIN.

When you specify UNMAP, you must also specify ID and AREA, and you may specify RETAIN. ID provides the address of the unique name that was returned by the IDENTIFY service, the same name that was input to the MAP service that created the window. AREA specifies the starting address of the virtual window. RETAIN specifies whether data from the object is to be retained or discarded. If you specify RETAIN=YES, the data from the object is retained in virtual storage. If you specify RETAIN=NO, the data is discarded from virtual storage, leaving the appearance of data that has been just obtained by a GETMAIN. RETAIN=NO is the default. UNMAP has no effect on the object itself and does not save data from the virtual window. If you want to save the data in the window, then invoke the SAVE before you invoke UNMAP.

UNACCESS
specifies that you are relinquishing your permission to read from or write to a data-in-virtual object. When you specify UNACCESS, you must also specify ID, which provides the address of the unique name that was returned by the IDENTIFY service. When UNACCESS is invoked, any outstanding windows for the specified ID are automatically unmapped with an implied RETAIN=NO.

UNIDENTIFY
specifies that the use of a data-in-virtual object under a previously assigned ID is ended. When you specify UNIDENTIFY, you must also specify ID, which provides the address of the unique name that was returned by the IDENTIFY service. If the object is still accessed or mapped under the specified ID, the system will automatically unaccess and unmap it with an implied RETAIN=NO.

,ID =*addr*
specifies the address of a field in storage where the IDENTIFY service stores a unique eight-byte name that it associates with the object. This name, which is like a token, is the output value of the IDENTIFY service; it is a required input value for all the other services.

,AREA =*address*
specifies the address of a four-byte field in storage containing a pointer to the start of the virtual window. The AREA parameter must be specified when you invoke the MAP and the UNMAP services. The starting address for an UNMAP request must be identical to the starting address of its corresponding MAP request. The virtual storage area that is occupied by a window must meet the following requirements:

- The window must begin on a 4096-byte (page) boundary and must be a multiple of 4096 bytes long.
- Virtual storage within the window must have been obtained by the GETMAIN macro from a single, pageable, private area subpool owned by the task that issued the IDENTIFY.
- The window cannot contain VIO storage.
- Pages within the window cannot be page fixed.

DDNAME =*addr*

specifies the address of a field containing the ddname for the object. The first byte of the field must be the number of characters in the ddname. The bytes following the first byte must contain the EBCDIC characters of the ddname itself. The ddname must conform to the standard syntax for ddnames. (One through eight alphameric or national characters, the first of which must be alphabetic or national.) DDNAME is required when you invoke IDENTIFY but it is not allowed when you invoke other services of the DIV macro.

LOCVIEW=MAP
LOCVIEW=NONE

LOCVIEW=MAP specifies that the data in the virtual window is to be protected from the unexpected changes that can occur when access to the object is not serialized. LOCVIEW=NONE specifies no protection, and is the default. LOCVIEW is valid only with ACCESS.

,MODE=READ
,MODE=UPDATE

specifies whether the object is being accessed for the purpose of reading or updating. If you are using the SAVE service to update an object, specify MODE=UPDATE. Otherwise, specify MODE=READ to signify read-only access to the object. MODE must be specified whenever you specify ACCESS.

,OFFSET =*addr*
,OFFSET=*

specifies the beginning of a continuous string of blocks in a data-in-virtual object. OFFSET is used with SPAN to define a continuous string of blocks in an object. OFFSET designates the location of the first block in the string, and SPAN designates how many blocks are in the string. An OFFSET value of zero designates the first block (the beginning) of an object. An OFFSET beyond the current end of the object is permitted as long as it remains within the maximum number of blocks allowed for the object and also within the absolute limit of (2**20)-1. if you omit offset or specify offset=*, a default OFFSET of zero is used. The OFFSET parameter can be specified with the MAP, RESET, and SAVE services.

,RETAIN=YES
,RETAIN=NO

specifies the retain mode of the window, which controls the actions of the MAP, UNMAP, and SAVE services. The retain mode determines what data appears in the window when the MAP service is invoked, and what data is left in virtual storage when UNMAP is invoked. It also affects how blocks are saved when you invoke the SAVE service, and how blocks are reset when you invoke RESET. The RETAIN parameter may be specified when you specify the MAP and the UNMAP parameters of the DIV macro. If the RETAIN parameter is not specified, the retain mode defaults to NO.

,SIZE =*addr*
,SIZE=*

specifies the address of a four-byte field where the system stores the size of the object. The size is stored in this field as a return value whenever you specify SAVE or ACCESS and also specify SIZE. When control is returned after the execution of a SAVE, the value that is returned is the minimum number of blocks that must be mapped to ensure that the entire object is mapped. If you omit SIZE or specify SIZE=*, then the size is not returned.

The size parameter may only be specified when you specify MAP, RESET, or SAVE.

,SPAN =*addr*
,SPAN=*

specifies the address of a four-byte field containing the number of blocks that are to be processed by the MAP, RESET, or SAVE services. These services operate only on a string of contiguous blocks. SPAN indicates how many blocks are in the string. It is used with OFFSET, which indicates the first block of the string.

For the RESET and SAVE services, the block string can include discontiguous mappings of an object. This lets you reset or save several maps in a single DIV macro invocation.

For the MAP service, the block string can extend beyond the end of the object, but it cannot extend beyond the maximum size allowed for the object. You can create a window that exceeds the size of the object. The maximum span allowed is (2**20)-1 bytes.

If you omit SPAN or specify SPAN=*, the SPAN default value is used. The default is also be used if the four-byte field contains zero. For the SAVE and RESET services, the default value is the number of blocks in the object from the specified or defaulted block to the end of the last mapped range. For the MAP service, the default is the current size of the object in blocks, minus the value specified by OFFSET. If the offset value is beyond the end of the object, the span defaults to one when you omit SPAN.

SPAN may be specified only when you specify MAP, RESET or SAVE.

,TYPE=DA

specifies that your program is using a data definition statement to identify the object. You must specify TYPE=DA whenever you specify IDENTIFY.

Return Codes, Reason Codes and Abend Codes.

If control is returned to the user after the DIV macro executes, a return code is supplied in the low order (rightmost) byte of general register 15. Successful completion is indicated by a zero return code and unsuccessful completion is indicated by a non-zero return code. When control is returned after an unsuccessful completion, a reason code is supplied in the two low order bytes of general register zero.

If control is not returned, an abend code and a reason code are supplied. The hexadecimal values of the reason, return and abend codes are:

Reason code	Return code	Abend code	Explanation
none	00	none	Successful completion.
0001	none	08B	Unknown service was requested.
0002	none	08B	Unknown parameter list format.
0003	none	08B	Input parameter list cannot be addressed.
0004	none	08B	Storage specified in the parameter list cannot be addressed.
0005	none	08B	The parameter list contains a reserved field that does not contain binary zeroes.
0006	none	08B	The caller is not running in task mode.
0007	none	08B	The caller is in cross memory mode.
0008	none	08B	An invalid TYPE is specified.
0009	none	08B	The supplied ID is not a valid ID or is an ID that cannot be used by the caller.
000A	08	none	There is another service currently executing with the specified ID.

Reason code	Return code	Abend code	Explanation
000B	none	08B	The object is already accessed with the specified ID.
000C	none	08B	The caller does not have proper RACF authorization to the requested object.
000D	none	08B	The requested window exceeds the maximum allowable size for the object.
000E	none	08B	The object is not currently accessed for the specified ID.
000F	none	08B	The specified range overlaps a range that is already mapped for the specified ID.
0010	none	08B	The specified range overlaps another mapped range.
0011	none	08B	Undetermined user error.
0012	none	08B	The virtual storage specified does not begin on a 4K boundary.
0013	none	08B	The virtual storage specified is not in a pageable private area subpool.
0014	none	08B	The virtual range specified cannot be used to map an object.
0015	none	08B	The virtual range specified contains at least one page that was not obtained by a GETMAIN macro, and RETAIN=NO was not specified.
0016	none	08B	The virtual range specified contains at least one fixed page.
0017	0C	none	An I/O error has occurred.
0018	none	08B	Caller does not have UPDATE access to the object.
0019	none	08B	A page to be saved or reset was in the page fixed state.
001A	04	none	The specified range does not encompass any mapped area of the object.
001B	none	08B	The virtual storage area specified to be unmapped is not currently mapped.
001C	08	none	The object cannot be accessed at the current time.
001D	none	08B	The accessed object is not at the correct Control Interval size.
001E	none	08B	The length of the ddname exceeds the maximum size allowed.
001F	none	08B	The caller's storage protect key is not the same as when IDENTIFY was invoked.
0020	none	08B	An ACCESS was attempted by a task that does not own the specified ID.
0021	0C	none	Portions of the object could not be retained in virtual storage as requested.
0022	none	08B	The specified storage to be mapped is not owned by the task chain that did the IDENTIFY.
0023	none	08B	Part or all of the specified storage to be mapped is not in the user's key.
0024	none	08B	The caller holds the local lock.
0025	none	08B	The caller is executing in an environment that precludes the use of SVCs.
0026	none	08B	The caller is not executing in 31 bit addressing mode.
0027	none	08B	The specified offset and span describe a range that goes beyond the maximum supported object size.
0028	08	none	The caller has attempted to access an empty object for reading.
0801	08	none	System error - Insufficient storage available to build the necessary data-in-virtual control block structure.
0802	08	none	System error - I/O driver failure.
0803	0C	none	System error - A necessary page table could not be read into real storage.
0804	0C	none	System error - Catalog update failed.
0805	none	08B	Unexpected system error
0806	0C	none	System error - I/O error.
0807	04	none	Media damage may be present in allocated DASD space. The damage is beyond the currently saved portion of the object. The SAVE completed successfully.
0808	08	none	System error - I/O from a previous request has not completed.

DIV (List Form)

The list form of the DIV macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DIV.
DIV	
b	One or more blanks must follow DIV.

	Valid parameters:
IDENTIFY	ID, TYPE, DDNAME
ACCESS	ID, MODE, SIZE, LOCVIEW
MAP	ID, AREA, OFFSET, SPAN, RETAIN
RESET	ID, OFFSET, SPAN
SAVE	ID, OFFSET, SPAN, SIZE
UNMAP	ID, AREA, RETAIN
UNACCESS	ID
UNIDENTIFY	ID
,AREA = *addr*	Initialized to zero if omitted *addr:* A-type address
,DDNAME = *addr*	Initialized to zero if omitted
,LOCVIEW = MAP ,LOCVIEW = NONE	**Default:** LOCVIEW = NONE
,ID = *addr*	Initialized to zero if omitted
,MODE = READ ,MODE = UPDATE	Initialized to zero if omitted
,TYPE = DA	Initialized to zero if omitted
,OFFSET = *addr* ,OFFSET = *	**Default:** OFFSET = 0
,RETAIN = YES ,RETAIN = NO	**Default:** RETAIN = NO
,SIZE = *addr* ,SIZE = *	See explanation of parameters if omitted
,SPAN = *addr* ,SPAN = *	See explanation of parameters if omitted
,MF = L	

,MF = L

specifies the list form of the DIV macro. The list form generates the DIV parameter list in line without any executable code or register usage.

DIV (Execute Form)

The execute form of the DIV macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DIV.
DIV	
b	One or more blanks must follow DIV.

	Valid parameters:
IDENTIFY	ID, TYPE, DDNAME
ACCESS	ID, MODE, SIZE, LOCVIEW
MAP	ID, AREA, OFFSET, SPAN, RETAIN
RESET	ID, OFFSET, SPAN
SAVE	ID, OFFSET, SPAN, SIZE
UNMAP	ID, AREA, RETAIN
UNACCESS	ID
UNIDENTIFY	ID
,AREA = *addr*	No change in executable parameter list if omitted *addr:* RX type address or register (2) - (12)
,DDNAME = *addr*	No change in executable parameter list if omitted
,LOCVIEW = MAP ,LOCVIEW = NONE	**Default:** LOCVIEW = NONE
,ID = *addr*	No change in executable parameter list if omitted
,MODE = READ ,MODE = UPDATE	No change in executable parameter list if omitted
,TYPE = DA	No change in executable parameter list if omitted
,OFFSET = *addr* ,OFFSET = *	**Default:** OFFSET = 0
,RETAIN = YES ,RETAIN = NO	**Default:** RETAIN = NO
,SIZE = *addr* ,SIZE = *	See explanation of parameters if omitted
,SPAN = *addr* ,SPAN = *	See explanation of parameters if omitted
,MF = (E,*addr*)	

,MF = (E,addr)
specifies the Execute form. In the Execute form, DIV will be called using the parameter list specified by "addr." "addr" indicates the address of the parameter list and may be (a) any address that is valid in an RX-type assembler language instruction or (b) one of the general registers 2 through 12 specified within parentheses. The register may be expressed either symbolically or as a decimal number. The specified parameter list will be updated for any parameters that are specified. Other parameter fields will be unaffected.

DIV (Modify Form)

The modify form of the DIV macro instruction is written as follows:

name	
b	One or more blanks must precede DIV.
DIV	
b	One or more blanks must follow DIV.

	Valid parameters:
IDENTIFY	ID, TYPE, DDNAME
ACCESS	ID, MODE, SIZE, LOCVIEW
MAP	ID, AREA, OFFSET, SPAN, RETAIN
RESET	ID, OFFSET, SPAN
SAVE	ID, OFFSET, SPAN, SIZE
UNMAP	ID, AREA, RETAIN
UNACCESS	ID
UNIDENTIFY	ID
,AREA=*addr*	No change in executable parameter list if omitted *addr:* RX type address or register (2) - (12)
,DDNAME=*addr*	No change in executable parameter list if omitted
,LOCVIEW=MAP ,LOCVIEW=NONE	**Default:** LOCVIEW=NONE
,ID=*addr*	No change in executable parameter list if omitted
,MODE=READ ,MODE=UPDATE	No change in executable parameter list if omitted
,TYPE=DA	No change in executable parameter list if omitted
,OFFSET=*addr* ,OFFSET=*	**Default:** OFFSET=0
,RETAIN=YES ,RETAIN=NO	**Default:** RETAIN=NO
,SIZE=*addr* ,SIZE=*	See explanation of parameters if omitted
,SPAN=*addr* ,SPAN=*	See explanation of parameters if omitted
,MF=(M,*addr*)	See explanation of parameters if omitted.

,MF=(M,*addr*)
specifies the MODIFY form. The modify form of the macro is used to modify an already defined DIV parameter list. It is exactly the same as the EXECUTE form except that DIV is not called. Registers 1 and 15 are destroyed.

DOM — Delete Operator Message

The DOM macro instruction deletes an operator message or group of messages from the display screen of the operator's console. It can also prevent messages from ever appearing on any operator's console. When a program no longer requires that a message be displayed, it can issue the DOM macro instruction to delete the message.

Depending on the timing of the DOM relative to the WTO(R), the message may or may not be displayed. If the message is being displayed, it is removed when space is required for other messages.

When a WTO or WTOR macro instruction is issued, the system assigns an identification number to the message and returns this number (24 bits right-justified) to the issuing program in general register 1. When the display of this message is no longer needed, you can issue the DOM macro instruction using the identification number that was returned in general register 1.

The DOM macro instruction is written as follows:

```
    name                    name: symbol. Begin name in column 1.
ƀ                           One or more blanks must precede DOM.
DOM
ƀ                           One or more blanks must follow DOM.

MSG=addr                    addr: register (1) - (12), or an address.
MSGLIST=list addr           list addr: symbol, RX-type address, or register (1) - (12).
TOKEN=addr                  addr: register (1) - (12), or an address.
    ,COUNT=addr             addr: register (2) - (12), or an address.
    ,REPLY=YES
```

The parameters are explained as follows:

MSG=
MSGLIST=
specifies the message numbers of messages to be deleted.

For MSG, the address or register contains the 24-bit, right-justified identification number of the message to be deleted. Use this parameter to delete a single message. If you use register 1, the macro expansion is shortened by two bytes.

For MSGLIST, the address is of a list of one or more fullwords, each word containing a 24-bit, right-justified identification number of a message to be deleted. A maximum of 60 identification numbers may be in the message list. If more than 60 identification numbers are in the list, only the first 60 are processed. Begin the list on a fullword boundary. When you are not using the COUNT parameter, indicate the end of the list by setting the high-order bit of the last fullword entry to 1. If you use register 1, the macro expansion is shortened by four bytes. If any register from 2 through 12 is used, the macro expansion is shortened by two bytes.

,COUNT=

The count field or register contains the one-byte count of 4-byte DOM ids associated with this request. The count value must be from 1 to 60. If this keyword is used, the issuer must not set the high order bit on in the last entry of the DOM parameter list. If this keyword is not specified, the DOM ids are treated as 3-byte ids. If an address is used instead of a register, the address points to a 1-byte field which contains the count. The COUNT keyword is invalid with the TOKEN keyword.

,REPLY=YES

specifies that the need for a reply to a WTOR message has been eliminated. REPLY=YES is invalid with TOKEN and COUNT. When you specify REPLY=YES, you must specify either MSG or MSGLIST to identify the message or group of messages that is to be deleted.

TOKEN=

The field or register contains the 4-byte TOKEN of the messages that are to be deleted. The messages that are deleted by TOKEN are the messages that were issued with this TOKEN via WTO. Unauthorized users may delete only those messages which were originally issued under the same jobstep TCB, ASID and system id. No DOM ids can be specified with this keyword. This keyword is mutually exclusive with the MSG, MSGLIST, and COUNT keywords.

Note: For any DOM keywords that allow a register specification, the value must be right-justified in the register and the remaining bytes within the register must be zero.

Example 1

Operation: Delete an operator message whose message id is in register 1.

```
DOM MSG=(1)
```

Example 2

Operation: Delete a list of operator messages, some of which may be WTORs.

```
DOM MSGLIST=ID2,REPLY=YES
```

Example 3

Operation: Delete a number of operator messages. The COUNT parameter indicates how many messages are to be deleted.

```
DOM MSGLIST=ID3,COUNT=COUNT4
```

Example 4

Operation: Delete all messages issued with a particular token.

```
DOM TOKEN=TOKEN1
```

ENQ — Request Control of a Serially Reusable Resource

ENQ assigns control of one or more serially reusable resources to the active task. If any of the resources are not available, the active task might be placed in a wait condition until all of the requested resources are available. Once control of a resource has been assigned to a task, it remains with that task until a DEQ macro instruction is issued or the task terminates. Register 15 is set to 0 if the request is satisfied.

You can also use ENQ to determine the status of the resource; whether it is immediately available or in use, and whether control has been previously requested for the active task in another ENQ macro instruction.

You can request either shared or exclusive use of a resource. The resource is represented in the ENQ by a pair of names, the *qname* and the *rname*, and a *scope* value. In order for ENQ/DEQ to coordinate the use of the resources:

- Everyone must use ENQ/DEQ.
- Everyone must use the same names and scope values for the same resources.
- Everyone must use consistent ENQ/DEQ protocol.

Issuing two ENQ macro instructions for one task for the same resource without an intervening DEQ macro instruction results in abnormal termination of the task, unless the second ENQ designates RET=TEST, USE, CHNG, or HAVE. If a task terminates while it still has control of any resources, all requests that this task made are automatically dequeued.

Global resource serialization counts and limits the number of concurrent resource requests from an address space. If an unconditional ENQ (an ENQ that uses the RET=NONE option) causes the count of concurrent resource requests to exceed the limit, the caller is abended with a system code of X'538'. See "Limiting Concurrent Requests for Resources" in Part I.

Note: When global resource serialization is active, the SYSTEM inclusion resource name list and the SYSTEMS exclusion resource name list are searched for every resource specified with a scope of SYSTEM or SYSTEMS. A resource whose name appears in one of these resource name lists might have its scope changed from the scope that appears on the macro instruction. See *Planning: Global Resource Serialization* for more information.

The standard form of the ENQ macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede ENQ.
ENQ	
ƀ	One or more blanks must follow ENQ.
(	
qname addr	*qname addr:* A-type address, or register (2) - (12).
,*rname addr*	*rname addr:* A-type address, or register (2) - (12).
,	**Default:** E
,E	
,S	
,	*rname length:* symbol, decimal digit, or register (2) - (12).
,*rname length*	
	Default: assembled length of *rname*
,	**Default:** STEP
,STEP	
,SYSTEM	
,SYSTEMS	
)	
,RET=CHNG	**Default:** RET=NONE
,RET=HAVE	
,RET=TEST	
,RET=USE	
,RET=NONE	
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

(
: specifies the beginning of the resource(s) description.

qname addr
: specifies the address in virtual storage of an 8-character name. The name can contain any valid hexadecimal character. Every program issuing a request for a serially reusable resource must use the same *qname*, *rname*, and scope to represent the resource. (See the section "Naming the Resource" for restrictions on naming qname.)

,*rname addr*
: specifies the address in virtual storage of the name used in conjunction with *qname* to represent a single resource. The name must be from 1 to 255 bytes long and can contain any valid hexadecimal characters.

,
,E
,S
: specifies whether the request is for exclusive (E) or shared (S) control of the resource. If the resource is modified while under control of the task, the request must be for exclusive control; if the resource is not modified, the request should be for shared control.

,

,*rname length*

specifies the length of the *rname* described above. If this parameter is omitted, the assembled length of the *rname* is used. You can specify a value between 1 and 255 to override the assembled length, or you may specify a value of 0. If 0 is specified, the length of the *rname* must be contained in the first byte at the *rname addr* specified above. This *rname length* parameter may be specified as an explicit constant (decimal digit), a label from an EQU assembler instruction (symbol), or a register (2)-(12). The *rname length* must be specified if either the name specified in the *rname* field, or the length attribute of the *rname* is defined by an EQU assembler instruction. Additionally, the *rname length* must be coded if the *rname addr* field is coded as a register.

,

,STEP
,SYSTEM
,SYSTEMS

specifies the scope of the resource used only within an address space (STEP), used by programs of more than one address space (SYSTEM), or shared between systems (SYSTEMS). If STEP is specified, a request for the same *qname* and *rname* from a program in another address space denotes a different resource. If SYSTEM or SYSTEMS is specified, requests for the same *qname*, *rname*, and scope from programs of any address space denote the same resource.

STEP, SYSTEM, and SYSTEMS are mutually exclusive and do not refer to the same resource. If two macro instructions specify the same *qname* and *rname*, but one specifies STEP and the other specifies SYSTEM or SYSTEMS, they are treated as requests for different resources.

Note: The SYSTEM option is not the same as the SYSTEMS option. When you specify SYSTEMS, you are spreading the scope of the resource across two or more processors. SYSTEM confines the scope to a single processor, but it includes two or more address spaces in the scope of the resource.

)

specifies the end of the resource(s) description.

Note: The parameters *qname addr*, *rname addr*, type of control, *rname length*, and the scope can be repeated within a single set of parentheses to indicate multiple resources. These parameters can be repeated until there is a maximum of 255 characters including the parentheses.

,RET=CHNG
,RET=HAVE
,RET=TEST
,RET=USE
,RET=NONE

specifies the type of request for all of the resources named above.

CHNG - the status of the resource specified is to be changed from shared to exclusive control.

HAVE - control of the resources is requested conditionally; that is, control is requested only if a request has not been made previously for the same task.

TEST - the availability of the resources is to be tested, but control of the resources is not requested.

USE - control of the resources is to be assigned to the active task only if the resources are immediately available. If any of the resources are not available, the active task is not placed in a wait condition.

NONE - control of all the resources is unconditionally requested.

See "Return Codes" for an explanation of the return codes for these requests.

,RELATED=*value*

specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
ENQUEUE  ENQ    (MAJOR,MINOR,S,8,STEP),                    X
                RELATED=(DEQUEUE,'OBTAIN RESOURCE')
   .
   .
   .
DEQUEUE  DEQ    (MAJOR,MINOR,8,STEP),                      X
                RELATED=(ENQUEUE,'RELEASE RESOURCE')
```

Return Codes

Return codes are provided by the control program only if you specify RET=TEST, RET=USE, RET=CHNG, or RET=HAVE; otherwise, return of the task to the active condition indicates that control of the resource has been assigned (or previously assigned) to the task. If all return codes for the resources named in the ENQ macro instruction are 0, register 15 contains 0. If any of the return codes are not 0, register 15 contains the address of a storage area containing the return codes, as shown in Figure 52. The return codes are placed in the parameter list resulting from the macro expansion in the same sequence as the resource names in the ENQ macro instruction. The return codes are shown in Figure 53.

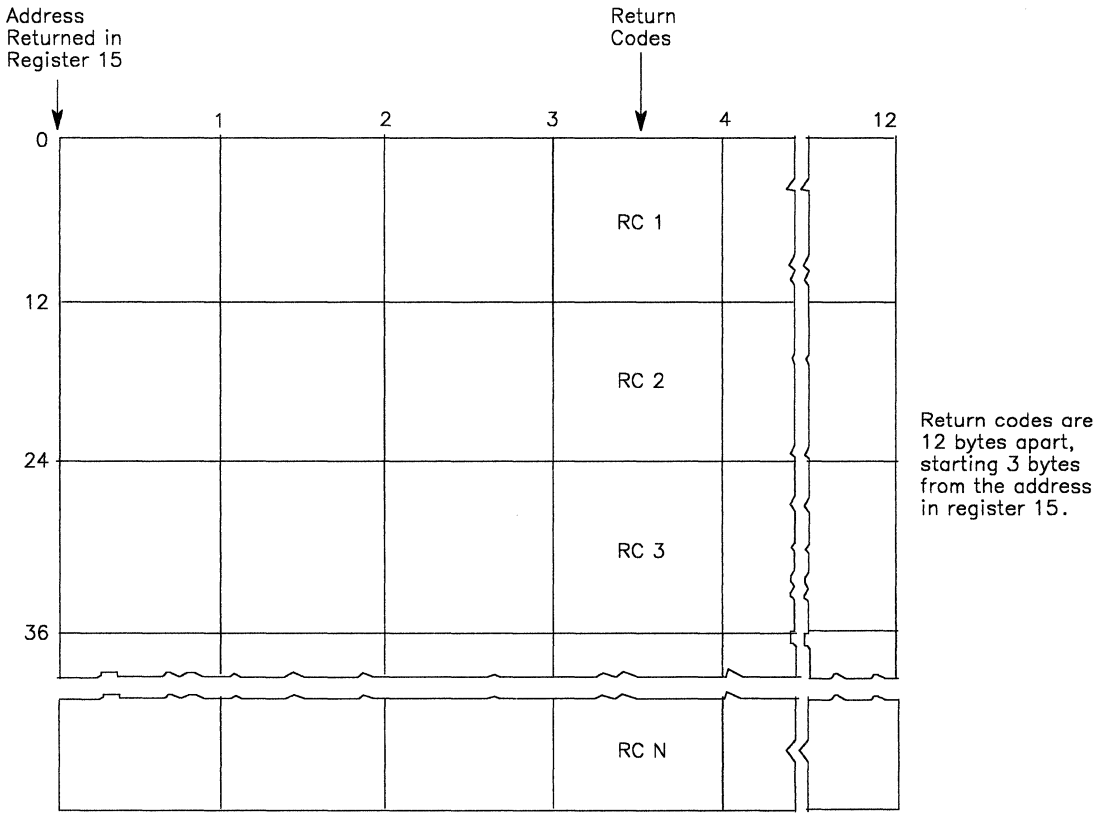

Figure 52. Return Code Area Used by ENQ

Hexadecimal Code	Meaning
0	For RET=TEST, the resource was immediately available. For RET=USE or RET=HAVE, control of the resource has been assigned to the active task. For RET=CHNG, the status of the resource has been changed to exclusive.
4	For RET=TEST or RET=USE, the resource is not immediately available. For RET=CHNG, the status cannot be changed to exclusive.
8	For RET=TEST, RET=USE, or RET=HAVE, this task has made a previous request for control of the same resource and this task has control of the resource. If bit 3 of the first byte of this entry in the ENQ parameter list is on, this task has shared control of the resource; if bit 3 is off, this task has exclusive control. For RET=CHNG, the resource was not queued or was not previously requested by the requesting task.
14	This task has made a previous request for control of the same resource, and this task does not have control of resource.
18	For RET=HAVE or RET=USE, the limit for the number of concurrent resource requests has been reached. The task does not have control of the resource unless some previous ENQ request caused the task to obtain control of the resource.

Figure 53. ENQ Return Codes

Example 1

Operation: Conditionally request shared control of a serially reusable resource that is known only within the address space (STEP). The resource is only to be obtained if immediately available. The resource will be used for read-only purposes. The length of *rname* is allowed to default.

```
ENQ       (MAJOR1,MINOR1,S,,STEP),RET=USE
```

Example 2

Operation: Unconditionally request exclusive control of 3 resources. The scope of each resource differs (STEP, SYSTEM, and SYSTEMS respectively). The *rname* length of the first resource is 3 characters and the *rname* length of the third resource is 8 characters. Allow the *rname* length of the second resource to default to its assembled length.

```
ENQ       (MAJOR4,MINOR4,E,3,,MAJOR2,MINOR2,,,SYSTEM,     X
          MAJOR3,MINOR3,E,8,SYSTEMS)
```

ENQ (List Form)

Use the list form of ENQ to construct a control program parameter list. Any number of resources can be specified in the ENQ macro instruction; therefore, the number of *qname*, *rname*, and scope combinations in the list form the ENQ macro instruction must be equal to the maximum number of *qname*, *rname*, and scope combinations in any execute form of the macro instruction that refers to that list form.

The list form of the ENQ macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ENQ.
ENQ	
b	One or more blanks must follow ENQ.
(	
qname addr	*qname addr:* A-type address.
, *rname addr*	*rname addr:* A-type address.
, ,E ,S	**Default:** E
, ,*rname length*	*rname length:* symbol or decimal digit. **Default:** assembled length of *rname*
, ,STEP ,SYSTEM ,SYSTEMS	**Default:** STEP
)	
,RET=CHNG ,RET=HAVE ,RET=TEST ,RET=USE ,RET=NONE	**Default:** RET=NONE
,RELATED=*value*	*value:* any valid macro keyword specification.
,MF=L	

The parameters are explained under the standard form of the ENQ macro instruction, with the following exception:

,MF=L
specifies the list form of the ENQ macro instruction.

ENQ (Execute Form)

A remote control program parameter list is used in, and can be modified by, the execute form of the ENQ macro instruction. The parameter list can be generated by the list form of ENQ.

The execute form of the ENQ macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ENQ.
ENQ	
b	One or more blanks must follow ENQ.
(	**Note:** (and) are the beginning and end of a parameter list. The entire list is optional. If nothing in the list is desired, then (,), and all parameters between (and) should not be specified. If something in the list is desired, then (,), and all parameters in the list should be specified as indicated at the left.
qname addr	*qname addr:* RX-type address, or register (2) - (12).
, *rname addr*	*rname addr:* RX-type address, or register (2) - (12).
, ,E ,S	**Default:** E
, ,*rname length*	*rname length:* symbol, decimal digit, or register (2) - (12).
, ,STEP ,SYSTEM ,SYSTEMS	**Default:** STEP
)	**Note:** See note opposite (above.
,RET=CHNG ,RET=HAVE ,RET=TEST ,RET=USE ,RET=NONE	**Default:** RET=NONE
,RELATED=*value*	*value:* any valid macro keyword specification.
,MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) - (12).

The parameters are explained under the standard form of the ENQ macro instruction, with the following exception:

,MF=(E,*ctrl addr*)
specifies the execute form of the ENQ macro instruction using a remote control program parameter list.

ESPIE — Extended SPIE

The ESPIE macro instruction extends the function of the SPIE (specify program interruption exits) macro instruction to callers in 31-bit addressing mode. Callers in either 24-bit or 31-bit addressing mode can issue the ESPIE macro instruction. Only callers in 24-bit addressing mode can issue the SPIE macro instruction. For additional information concerning the relationship between the SPIE and the ESPIE macro instructions, see the section "Interruption Services" in Part I.

The ESPIE macro instruction performs the following functions using the options specified:

- Establishes an ESPIE environment (that is, identifies the interruption types that are to cause entry to the ESPIE exit routine) by executing the SET option of the ESPIE macro instruction.
- Deletes an ESPIE environment (that is, cancels the current SPIE/ESPIE environment) by executing the RESET option of the ESPIE macro instruction.
- Determines the current SPIE/ESPIE environment by executing the TEST option of the ESPIE macro instruction.

SET Option

The SET option of the ESPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ESPIE.
ESPIE	
b	One or more blanks must follow ESPIE.
SET	
,*exit addr*	*exit addr:* A-type address, or register (2) - (12).
,(*interruptions*)	*interruptions:* decimal digits 1-15 and expressed as: single values: (2, 3, 4, 7, 8, 9, 10) ranges of values: ((2, 4), (7, 10)) combinations: (2, 3, 4, (7, 10))
,PARAM=*list addr*	*list addr:* A-type address or register (2) - (12).

The parameters are explained as follows:

SET
indicates that an ESPIE environment is to be established.

,exit addr
specifies the address of the exit routine to be given control when program interruptions of the type specified by *interruptions* occur. The exit routine receives control in the same addressing mode as the issuer of the ESPIE macro instruction.

,(interruptions)
indicates the interruption types that are being trapped. The interruption types are:

Number	Interruption Type
1	Operation
2	Privileged operation
3	Execute
4	Protection
5	Addressing
6	Specification
7	Data
8	Fixed-point overflow (maskable)
9	Fixed-point divide
10	Decimal overflow (maskable)
11	Decimal divide
12	Exponent overflow
13	Exponent underflow (maskable)
14	Significance (maskable)
15	Floating-point divide

These interruption types can be designated as one or more single numbers, as one or more pairs of numbers (designating ranges of values), or as any combination of the two forms. For example, (4,8) indicates interruption types 4 and 8; ((4,8)) indicates interruption types 4 through 8.

If a program interruption type is maskable, the corresponding program mask bit in the PSW is set to 1. If a maskable interruption is not specified, the corresponding bit in the PSW is set to 0. Interruption types not specified above (except for type 17, which is described in *SPL: System Macros and Facilities*) are handled by the control program. The control program forces an abend with the program check as the completion code. If an ESTAE-type recovery routine is also active, the SDWA indicates a system-forced abnormal termination. The registers at the time of the error are those of the control program.

Note: For both ESPIE and SPIE – If you are using vector instructions and an exception of 8, 12, 13, 14, or 15 occurs, your recovery routine can check the exception extension code (the first byte of the two-byte interruption code in the EPIE or PIE) to determine whether the exception was a vector or scalar type of exception.

For more information about the exception extension code, see *IBM System/370 Vector Operations.*

,PARAM=*list addr*
specifies the fullword-address of a parameter list that is to be passed by the caller to the exit routine.

On return from the SET option of the ESPIE macro instruction, the registers contain the following information:

Register	Content
0	Unpredictable
1	Token representing the previously active SPIE/ESPIE environment
2-13	Unchanged
14	Unpredictable
15	Return code of 0

Example 1

Operation: Give control to an exit routine for interruption types 1 and 4. EXIT is the location of the exit routine to be given control and PARMLIST is the location of the user parameter list to be used by the exit routine.

```
ESPIE SET,EXIT,(1,4),PARAM=PARMLIST
```

RESET Option

The RESET option of the ESPIE macro cancels the current SPIE/ESPIE environment and re-establishes the previously active SPIE/ESPIE environment identified by the token specified.

The RESET option of the ESPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede ESPIE.
ESPIE	
ƀ	One or more blanks must follow ESPIE.
RESET	
,token	*token:* RX-type address, or register (1), (2) - (12).

The parameters are explained as follows:

RESET
indicates that the current ESPIE environment is to be deleted and the previously active SPIE/ESPIE environment specified by *token* is to be re-established.

,token
specifies a fullword that contains a token representing the previously active SPIE/ESPIE environment. This is the same token that ESPIE processing returned to the caller when the ESPIE environment was established using the SET option of the ESPIE macro instruction.

If the token is zero, all SPIEs and ESPIEs are deleted.

On return from ESPIE RESET, the contents of the registers are as follows:

Register	Contents
0	Unpredictable
1	Token identifying the new active SPIE/ESPIE environment
2-13	Unchanged
14	Unpredictable
15	Return code of 0

Example 1

Operation: Cancel the current SPIE/ESPIE environment and restore the SPIE/ESPIE environment represented by the contents of TOKEN.

```
ESPIE RESET,TOKEN
```

TEST Option

The TEST option of the ESPIE macro instruction determines the active SPIE/ESPIE environment and returns the information in a four-byte parameter list.

The TEST option of the ESPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede ESPIE.
ESPIE	
ƀ	One or more blanks must follow ESPIE.
TEST	
,*parm addr*	*parm addr:* RX-type address, or register (1), (2) - (12).

The parameters are explained as follows:

TEST
indicates a request for information concerning the active or current SPIE/ESPIE environment. ESPIE processing returns this information to the caller in a four-word parameter list located at *parm addr*.

,parm addr
specifies the address of a four-word parameter list aligned on a fullword boundary. The parameter list has the following form:

Word	Content
0	Address of the user-exit routine (31-bit address with the high-order bit set to 0)
1	Address of the user-defined parameter list
2	Mask of program interruption types (Note: Bit 1 corresponds to interrupt code 1, bit 2 to interrupt code 2, and so on.)
3	Zero

On return from ESPIE TEST, the registers contain the following information:

Register	Contents
0	Unpredictable
1-13	Unchanged
14	Unpredictable
15	Return code as follows:

Code	Meaning
0	An ESPIE exit is active and the four-word parameter list contains the information described under the *parm addr* parameter of the ESPIE macro instruction.
4	A SPIE exit is active and word 1 of the parameter list contains the address of the current PICA. Words 0, 2, and 3 of the parameter list are unpredictable.
8	A SPIE or ESPIE exit is not active. All the words of the parameter list are unpredictable.

Example 1

Operation: Identify the active SPIE/ESPIE environment. Return the information about the exit routine in the four-word parameter list, PARMLIST. Also return, in register 15, an indicator of whether a SPIE, ESPIE, or neither is active.

```
ESPIE TEST,PARMLIST
```

ESPIE (List Form)

The list form of the ESPIE macro instruction builds a nonexecutable problem program parameter list that can be referred to or modified by the execute form of the ESPIE macro instruction.

The list form of the ESPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ESPIE.
ESPIE	
b	One or more blanks must follow ESPIE.
SET	
,*exit addr*	*exit addr:* A-type address. **Note:** This parameter must be specified on either the list or the execute form of the macro instruction.
,(*interruptions*)	*interruptions:* decimal digit 1-15 and expressed as: single values: (2, 3, 4, 7, 8, 9, 10) range of values: ((2, 4), (7, 10)) combinations: (2, 3, 4, (7, 10))
,PARAM=*list addr*	*list addr:* A-type address.
,MF=L	

The parameters are explained under the standard form of the ESPIE macro instruction with the following exception:

,MF=L
specifies the list form of the ESPIE macro instruction.

Example 1

Operation: Build a nonexecutable problem program parameter list that will cause control to be transferred to the exit routine, EXIT for the interruption types specified in the execute form of the macro instruction. Provide the address of the user parameter list, PARMLIST.

```
LIST1 ESPIE SET,EXIT,,PARAM=PARMLIST,MF=L
```

ESPIE (Execute Form)

The execute form of the ESPIE macro instruction can refer to and modify the parameter list constructed by the list form of the ESPIE macro instruction.

The execute form of the ESPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ESPIE.
ESPIE	
b	One or more blanks must follow ESPIE.
SET	
,*exit addr*	*exit addr:* RX-type address or register (2) - (12). **Note:** This parameter must be specified on either the list or the execute form of the macro instruction.
,(*interruptions*)	*interruptions:* decimal digit 1-15 and expressed as: single values: (2, 3, 4, 7, 8, 9, 10) range of values: ((2, 4), (7, 10)) combinations: (2, 3, 4, (7, 10))
,PARAM=*list addr*	*list addr:* RX-type address or register (2) - (12).
,MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1), (2) - (12).

The parameters are explained under the standard form of the ESPIE macro instruction with the following exception:

,MF=(E,ctrl addr)
specifies the execute form of the ESPIE macro instruction using a remote control program parameter list.

Example 1

Operation: Give control to a user exit routine for interruption types 1, 4, 6, 7, and 8. The exit routine address and the address of a user parameter list for the exit routine are provided in a remote control program parameter list at LIST1.

```
ESPIE SET,,(1,4,(6,8)),MF=(E,LIST1)
```

ESTAE — Extended Specify Task Abnormal Exit

The ESTAE macro instruction provides recovery capability facilities. Issuance of the ESTAE macro instruction or ATTACH with the STAI (specify task abnormal interrupts) or ESTAI (extended STAI) option allows the user to intercept a scheduled ABEND. Control is given to a user specified exit routine in which the user may perform pre-termination processing, diagnose the cause of ABEND, and specify a retry address if he wishes to avoid the termination. These exits operate in both problem program and supervisor state.

This macro can be assembled compatibly between MVS/XA and MVS/370 through the use of the SPLEVEL macro instruction. Default processing will result in an expansion of the macro that operates only with MVS/XA. See the topic "Selecting the Macro Level" for additional information.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction. ESTAE exits and retry routines execute in the same address mode as the program that issues the ESTAE macro instruction.

The standard form of the ESTAE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ESTAE.
ESTAE	
b	One or more blanks must follow ESTAE.
exit addr 0	*exit addr:* A-type address, or register (2) - (12).
,CT ,OV	**Default:** CT
,PARAM=*list addr*	*list addr:* A-type address, or register (2) - (12).
,XCTL=NO ,XCTL=YES	**Default:** XCTL=NO
,PURGE=NONE ,PURGE=QUIESCE ,PURGE=HALT	**Default:** PURGE=NONE
,ASYNCH=YES ,ASYNCH=NO	**Default:** ASYNCH=YES
,TERM=NO ,TERM=YES	**Default:** TERM=NO
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

exit addr
0

specifies the address of an ESTAE exit routine to be entered if the task issuing this macro instruction terminates abnormally. If 0 is specified, the most recent ESTAE exit is deleted.

,CT
,OV

specifies the creation of a new ESTAE exit (CT) or indicates that parameters passed in this ESTAE macro instruction are to overlay the data contained in the previous ESTAE exit (OV).

,PARAM=*list addr*

specifies the address of a user-defined parameter list containing data to be used by the ESTAE exit routine when it is scheduled for execution.

,XCTL=NO
,XCTL=YES

specifies that the ESTAE macro instruction will be deleted (NO) or will not be deleted (YES) if an XCTL macro instruction is issued by this program.

,PURGE=NONE
,PURGE=QUIESCE
,PURGE=HALT

specifies that all outstanding requests for I/O operations will not be saved when the ESTAE exit is taken (HALT), that I/O processing will be allowed to continue normally when the ESTAE exit is taken (NONE), or that all outstanding requests for I/O operations will be saved when the ESTAE exit is taken (QUIESCE). If QUIESCE is specified, the user's retry routine can restore the outstanding I/O requests.

Note: If PURGE=NONE is specified, all control blocks affected by input/output processing may continue to change during ESTAE exit routine processing.

If PURGE=NONE is specified and the ABEND was originally scheduled because of an error in input/output processing, an ABEND recursion will develop when an input/output interruption occurs, even if the exit routine is in progress. Thus, it will appear that the exit routine failed when, in reality, input/output processing was the cause of the failure.

Do not use PURGE=HALT to stop processing a data set if you expect to continue reading the data set at a different point.

Notes:

1. *You should understand PURGE processing before using this parameter. For information on PURGE processing, see System-Data Administration.*

2. *If you specify PURGE=HALT or PURGE=QUIESCE, but I/O is not restored,*

   - **While using SAM or ISAM,** *only the input/output event on which the purge is done will be posted. Subsequent event control blocks (ECBs) will not be posted. If you issue further data management macros, such as GET/PUT, READ/WRITE or CLOSE, after a PURGE is issued during ESTAE recovery, a wait may occur in an access method module.*

   - **While using ISAM,**

     – *The ISAM check routine will treat purged I/O as normal I/O.*

     – *Part of the data set might be destroyed if the data set was being updated or added to when the failure occurred.*

,ASYNCH=YES
,ASYNCH=NO

specifies that asynchronous exit processing will be allowed (YES) or prohibited (NO) while the user's ESTAE exit is executing.

ASYNCH=YES must be coded if:

- Any supervisor services that require asynchronous interruptions to complete their normal processing are going to be requested by the ESTAE exit routine.
- PURGE=QUIESCE is specified for any access method that requires asynchronous interruptions to complete normal input/output processing.
- PURGE=NONE is specified and the CHECK macro instruction is issued in the ESTAE exit routine for any access method that requires asynchronous interruptions to complete normal input/output processing.

Note: If ASYNCH=YES is specified and the ABEND was originally scheduled because of an error in asynchronous exit handling, an ABEND recursion will develop when an asynchronous exit handling was the cause of the failure.

,TERM=NO
,TERM=YES

specifies that the exit routine associated with the ESTAE request will be scheduled (YES) or will not be scheduled (NO), in addition to normal ESTAE processing, in the following situations:

- Cancel by operator.
- Forced logoff.
- Expiration of job step timer.
- Exceeding of wait time limit for job step.
- ABEND condition because of DETACH of an incomplete subtask when the STAE option was not specified on the DETACH.
- ABEND of the attaching task when the ESTAE macro instruction was issued by a subtask.
- ABEND of job step task when a non-job step task requested ABEND with the STEP option.

When the exit routine is entered because of one of the preceding reasons, retry will not be permitted. If dump is requested at the time of ABEND, it is taken prior to entry into the exits.

Note: If DETACH was issued with the STAE parameter, the following will occur for the task to be detached:

- All ESTAE exits will be entered.
- The most recently established STAE exit will be entered.
- All STAI/ESTAI exits will be entered unless return code 16 is returned from one of the STAI exits.

In these cases, entry to the exit is prior to dumping and retry will not be permitted.

,RELATED=*value*
specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
DEFESTAE ESTAE (4),CT,PARAM=(2),RELATED=(DELESTAE,
               'DELETE ESTAE')
   .
   .
   .
DELESTAE ESTAE 0,RELATED=(DEFESTAE,'DEFINE ESTAE')
```

Note: This macro instruction will fit on one line when coded, so there is no need for a continuation indicator.

Control is returned to the instruction following the ESTAE macro instruction. When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion of ESTAE request.
04	ESTAE OV was specified with a valid exit address, but the current exit is either nonexistent, not owned by the user's RB, or is not an ESTAE exit.
0C	Delete (an exit address equal to zero) was specified and either there are no exits for this TCB, the most recent exit is not owned by the caller, or the most recent exit is not as ESTAE exit.
10	An unexpected error was encountered while processing this request.
14	ESTAE was unable to obtain storage for an SCB.

Example 1

Operation: Request an overlay of the existing ESTAE recovery exit (at ADDR), with the following options: parameter list is as PLIST, I/O will be halted, no asynchronous exits will be taken, ownership will be transferred to the new request block resulting from any XCTL macro instructions.

```
ESTAE   ADDR,OV,PARAM=PLIST,XCTL=YES,PURGE=HALT,ASYNCH=NO
```

Example 2

Operation: Provide the pointer to the recovery code in the register called EXITPTR, and the address of the ESTAE exit parameter list in register 9. Register 8 points to the area where the ESTAE parameter list (created with the MF=L option) was moved.

```
ESTAE   (EXITPTR),PARAM=(9),MF=(E,(8))
```

ESTAE (List Form)

The list form of the ESTAE macro instruction is used to construct a remote control program parameter list.

The list form of the ESTAE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ESTAE.
ESTAE	
b	One or more blanks must follow ESTAE.

exit addr 0	*exit addr:* A-type address.
,PARAM=*list addr*	*list addr:* A-type address.
,PURGE=NONE ,PURGE=QUIESCE ,PURGE=HALT	**Default:** PURGE=NONE
,ASYNCH=YES ,ASYNCH=NO	**Default:** ASYNCH=YES
,TERM=NO ,TERM=YES	**Default:** TERM=NO
,RELATED=*value*	*value:* any valid macro keyword specification.
,MF=L	

The parameters are explained under the standard form of the ESTAE macro instruction, with the following exception:

,MF=L
specifies the list form of the ESTAE macro instruction.

ESTAE (Execute Form)

A remote control program parameter list is used in, and can be modified by, the execute form of the ESTAE macro instruction. The control program parameter list can be generated by the list form of the ESTAE macro instruction. If the user desires to dynamically change the contents of the remote ESTAE parameter list, he may do so by coding a new exit address and/or a new parameter list address. If exit address or PARAM is coded, only the associated field in the remote ESTAE parameter list will be changed. The other field will remain as it was before the current ESTAE request was made.

The execute form of the ESTAE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede ESTAE.
ESTAE	
b	One or more blanks must follow ESTAE.
exit addr 0	*exit addr:* RX-type address, or register (2) - (12).
,CT ,CV	
,PARAM=*list addr*	*list addr:* RX-type address, or register (2) - (12).
,XCTL=NO ,XCTL=YES	
,PURGE=NONE ,PURGE=QUIESCE ,PURGE=HALT	
,ASYNCH=YES ,ASYNCH=NO	
,TERM=NO ,TERM=YES	
,RELATED=*value*	*value:* any valid macro keyword specification.
,MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) or (2) - (12).

The parameters are explained under the standard form of the ESTAE macro instruction, with the following exception:

,MF=(E,*ctrl addr*)
specifies the execute form of the ESTAE macro instruction using a remote control program parameter list.

EVENTS — Wait for One or More Events to Complete

The EVENTS macro instruction is the same as the WAIT ECBLIST macro, with one additional function: it notifies the calling program that event control blocks (ECBs) have completed and the order in which they completed.

The macro performs the following functions:

- Creates and deletes EVENTS tables.
- Initializes and maintains a list of completed event control blocks.
- Provides for single or multiple ECB processing.

For a detailed explanation of how to use EVENTS to perform these functions see "Using the EVENTS Macro Instruction" in this section.

This macro can be assembled compatibly between MVS/XA and MVS/370 through the use of the SPLEVEL macro instruction. Default processing will result in an expansion of the macro that operates only with MVS/XA. See the topic "Selecting the Macro Level" for additional information.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction.

The EVENTS macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede EVENTS.
EVENTS	
b	One or more blanks must follow EVENTS.

ENTRIES=*n*	*n:* variable or decimal digit 1-32,767.
ENTRIES=*addr*	*addr:* register (2) - (12).
ENTRIES=DEL,TABLE=*table address*	*table address:* symbol, RX-type address, or register (2) - (12).
TABLE=*table address*	**Note:** If ENTRIES=*n* or ENTRIES=DEL, TABLE=*table address* is not specified, no other parameter should be specified.
,WAIT=NO ,WAIT=YES	**Default:** None.
,ECB=*ecb address*	*ecb address:* symbol, RX-type address, or register (2) - (12).
,LAST=*last address*	*last address:* symbol, RX-type address, or register (2) - (12).
	Note: Optional parameters are only valid when TABLE=*table address* is the only required parameter specified.

The parameters are explained as follows:

ENTRIES=*n*

specifies either a register or a decimal number from 1 to 32,767 that specifies the maximum number of completed ECB addresses that can be processed concurrently in an EVENTS table.

Note: When this parameter is specified, no other parameter should be specified.

ENTRIES=DEL,TABLE= *table address*
specifies that the EVENTS table whose address is specified by TABLE=*table address* is to be deleted. The user is responsible for deleting all of the tables he creates; however, all existing tables are automatically freed at task termination.

Notes:

1. *When this parameter is specified no other parameter should be specified.*
2. *table address specifies a storage location below 16 megabytes.*

TABLE= *table address*
specifies either a register number or the address of a word containing the address of the EVENTS table associated with the request. The address specified with the operand TABLE must be that of an EVENTS table created by this task.

Note: *table address* specifies a storage location below 16 megabytes.

,WAIT=NO
,WAIT=YES
specifies whether or not to put the issuing program in a wait state when there are no completed events in the EVENTS table (specified by the TABLE= parameter).

,ECB= *ecb address*
specifies either a register number or the address of a word containing the address of an event control block. The EVENTS macro instruction should be used to initialize any event-type ECB. To avoid the accidental destruction of bit settings by a system service such as an access method, the ECB should be initialized after the system service that will post the ECB has been initiated (thus making the ECB eligible for posting) and before the EVENTS macro is issued to wait on the EVENTS table.

Notes:

1. *Register 1 should not be specified for the ECB address.*
2. *This parameter may not be specified with the LAST= parameter.*
3. *If only ECB initialization is being requested, neither WAIT=NO nor WAIT=YES should be specified, to prevent any unnecessary WAIT processing from occurring.*

,LAST= *last address*
specifies either a register number or the address of a word containing the address of the last EVENT parameter list entry processed.

Notes:

1. *Register 1 should not be specified for the LAST address.*
2. *This parameter should not be specified with the ECB= parameter.*
3. *last address specifies a storage location below 16 megabytes.*

Using the EVENTS Macro Instruction

The following explains the different uses of EVENTS:

- Creating EVENTS Tables — When ENTRIES=n is specified, the system creates an EVENTS table with "n" entries for completed ECB addresses. This table is queued on the EVENTS table queue associated with the task. (There is no limit to the number of EVENTS tables that can be queued for a single task.) The address of the EVENTS table is returned to the user in register 1. See Figure 54.

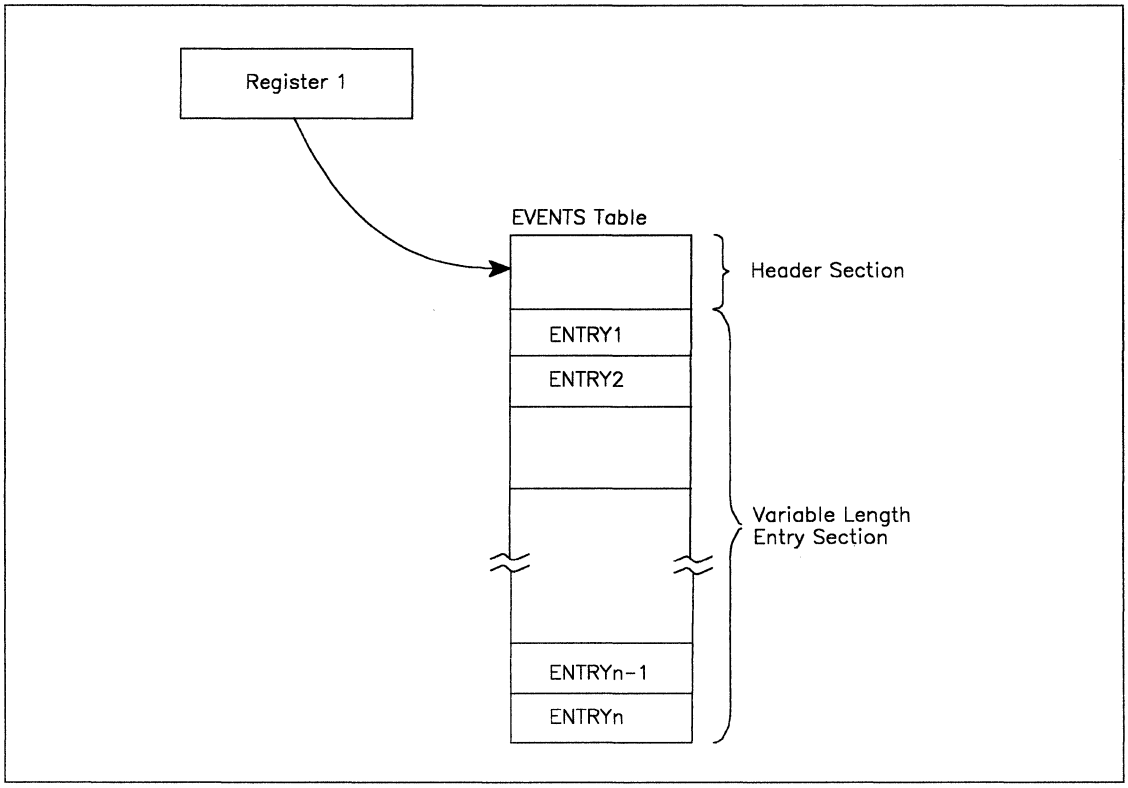

Figure 54. Creating a Table

- Deleting EVENTS Tables — When ENTRIES=DEL,TABLE=table address is specified, the EVENTS table whose address is specified by the TABLE=table address parameter shall be deleted. The address specified with the TABLE operand must be that of an EVENTS table created by this task. The user is responsible for deleting all of the tables he creates; however, all existing tables are automatically freed at task termination.

- Initializing ECBs — When an ECB is created, bits 0 (wait bit) and bit 1 (post bit) must be set to zero. When an EVENTS ECB= macro instruction is issued, bit 0 of the associated event control block is set to 1. When a POST macro instruction is issued, bit 1 of the associated event control block is set to 1 and bit 0 is set to 0. If the ECB is reused, bit 0 and bit 1 must be set to zero before either a WAIT, EVENTS ECB=, or POST macro instruction can be specified. If, however, the bits are set to zero before the ECB has been posted, any task waiting for that ECB to be posted will remain in wait state.

- Maintaining a List of Completed EVENT Control Blocks — After the ECB has been initialized the POST macro sets the complete bit and puts the address of the completed ECB in the EVENTS table.

- Providing Single or Multiple ECB Processing — When the WAIT parameter is specified and there are completed ECBs in the EVENTS table, the address of the parameter list is returned in register 1. The parameter list has the following format:

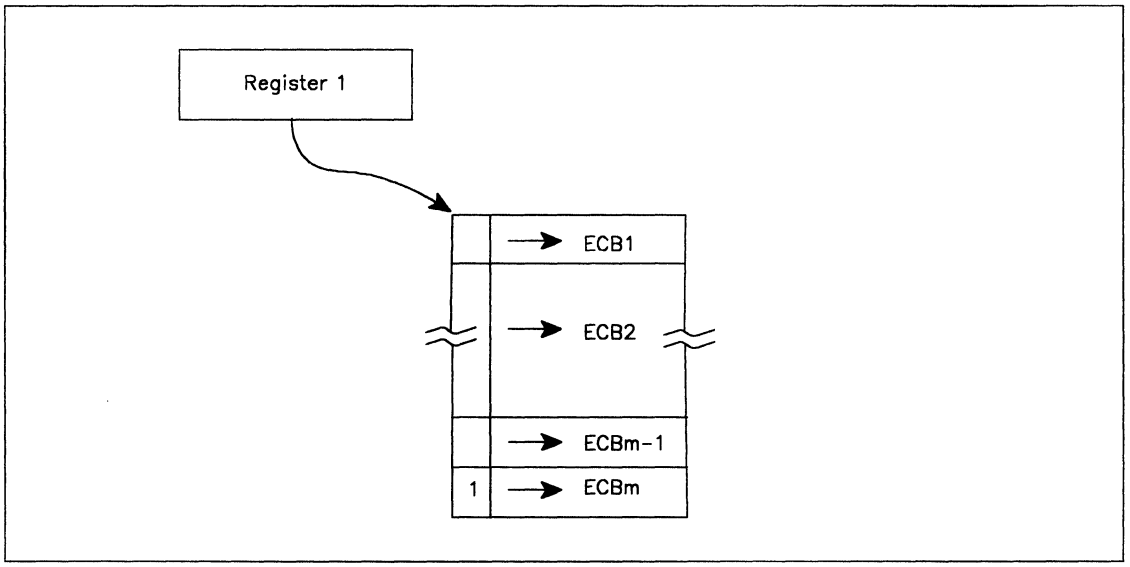

Figure 55. Parameter List Format

The parameter list contains completed ECB addresses in post occurrence order. The high order bit of the last word in the list is set to 1. The user may choose to process the entire list (see LAST parameter) or one event at a time by successive EVENTS requests with the WAIT= option.

However, if WAIT=NO is specified and no ECBs are posted in the EVENTS table, register 1 contains a zero when the user receives control.

When a user has processed more than one ECB in the parameter list returned from the previous EVENTS WAIT= macro, the LAST= parameter should be used to indicate the last ECB processed. The EVENTS macro removes from the parameter list all entries from the first thru the last specified by LAST, and then completes processing the request according to the WAIT= specification.

In the illustration that follows, ECBs 6 through 10 were posted to the parameter list while the user was processing 1 through 5.

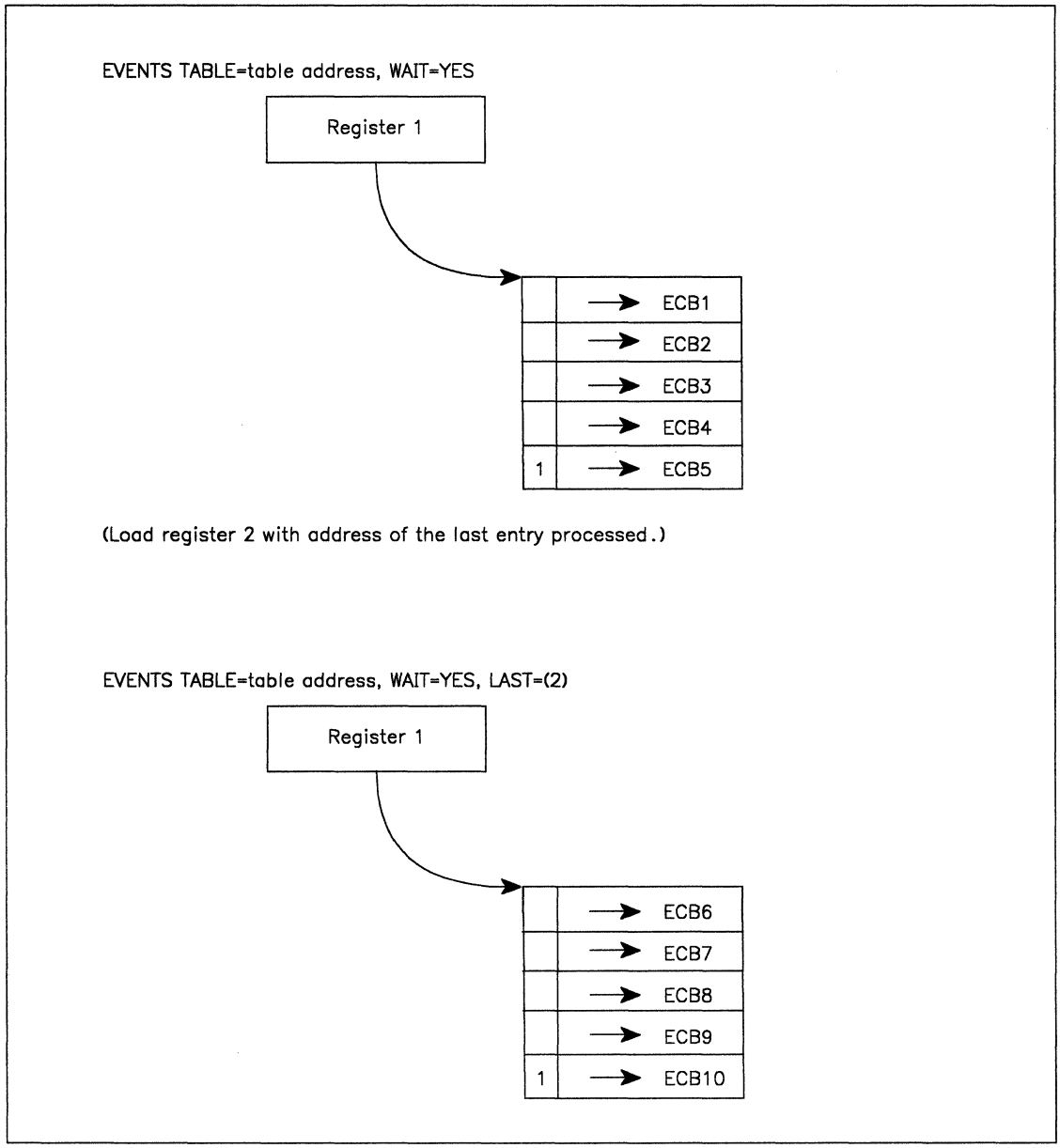

Figure 56. Posting the Parameter List

This figure demonstrates processing one event at a time.

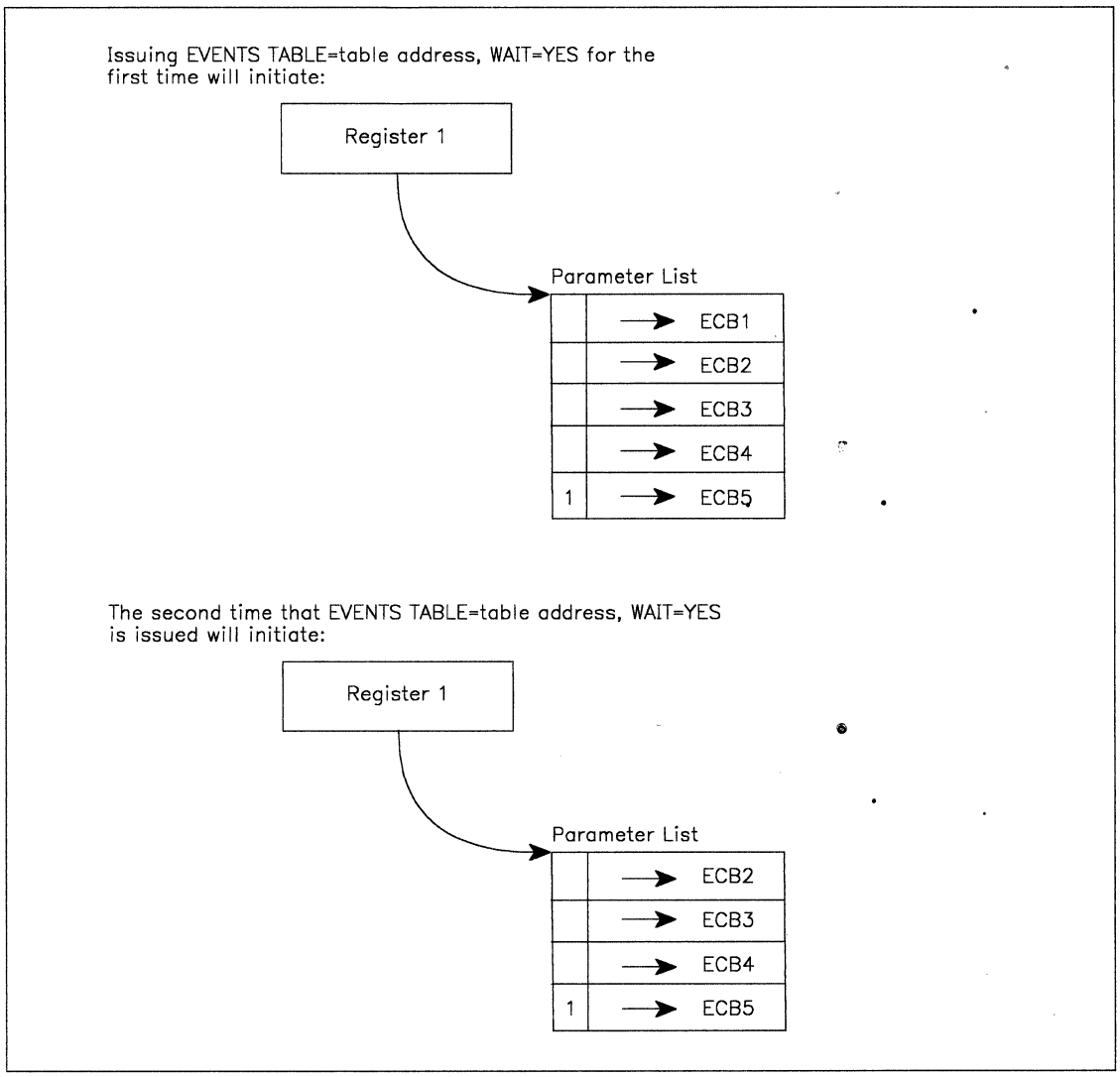

Figure 57. Processing One Event At a Time

Example 1

The following shows total processing via EVENTS.

EVENTS and ECB Initialization

```
        START
        EVENTS    ENTRIES=1000
        ST        R1,TABADD
        WRITE     ECBA
        LA        R2,ECBA
        EVENTS    TABLE=TABADD,ECB=(R2)
```

Parameter List Processing

```
        BEGIN
        EVENTS    TABLE=TABADD,WAIT=YES
        LR        R3,R1          PARMLIST ADDR
        B         LOOP2          GO TO PROCESS ECB
LOOP1   EVENTS    TABLE=TABADD,WAIT=YES,LAST=(R3)
        LR        R3,R1          SAVE POINTER
LOOP2   EQU       *              PROCESS COMPLETED EVENTS
        TM        0(R3),X'80'    TEST FOR MORE EVENTS
        BO        LOOP1          IF NONE, GO WAIT
        LA        R3,4(,R3)      GET NEXT ENTRY
        B         LOOP2          GO PROCESS NEXT ENTRY
```

Deleting EVENTS Table

```
        EVENTS    TABLE=TABADD,ENTRIES=DEL
TABADD  DS        F
```

Example 2

Processing One ECB at a Time.

```
                  EVENTS      ENTRIES=10
                  ST          1,TABLE
        NEXTREC   GET         TPDATA,KEY
                  ENQ         (RESOURCE,ELEMENT,E,,SYSTEM)
                  READ        DECBRW,KU,,'S',MF=E
                  LA          3,DECBRW
                  EVENTS      TABLE=TABLE,ECB=(3),WAIT=YES
                  WRITE       DECBRW,K,MF=E
                  LA          3,DECBRW
        RETEST    EVENTS      TABLE=TABLE,ECB=(3),WAIT=NO
                  LTR         1,1
                  BNZ         NEXTREC
                  B           RETEST
        TABLE     DS          F
```

FREEMAIN — Free Virtual Storage

The FREEMAIN macro instruction releases one or more areas of virtual storage, or an entire virtual storage subpool, previously assigned to the active task as a result of a GETMAIN macro instruction. The active task is abnormally terminated if the specified virtual storage does not start on a doubleword boundary or, for an unconditional request, if the specified area or subpool is not currently allocated to the active task. Register 15 is set to 0 to indicate successful completion. For a conditional FREEMAIN, register 15 is set to 4 if the specified area or subpool is not currently allocated to the active task.

In the parameters discussed below, EU, LU, and VU specify unconditional requests and result in the same processing as E, L, and V, respectively. The two formats for these requests are available to maintain compatibility with the GETMAIN formats.

The standard form of the FREEMAIN macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede FREEMAIN.
FREEMAIN	
b	One or more blanks must follow FREEMAIN.

LC,LA=*length addr* LU,LA=*length addr* L,LA=*length addr* VC VU V EC,LV=*length value* EU,LV=*length value* E,LV=*length value* RC,LV=*length value* RC,SP=*subpool nmbr* RU,LV=*length value* RU,SP=*subpool nmbr* R,LV=*length value* R,SP=*subpool nmbr*	*length addr:* A-type address, or register (2) - (12). *length value:* symbol, decimal digit, or register (2) - (12). If R, RC, or RU is specified, register (0) may also be specified. *subpool nmbr:* symbol, decimal digit 0-127, or register (2) - (12). If R is specified, register (0) may also be specified. **Note:** For subpool freemains, if the forms RC,SP=*subpool nmbr* or RU,SP=*subpool nmbr* or R,SP=*subpool nmbr* are specified, no other parameters except RELATED may be specified. SP= must be specified for subpool FREEMAINS; for other types of FREEMAIN, SP= is optional and defaults to SP=0. **Note:** RC and RU are the only parameters that can be used to free storage above 16 Mb.
,A=*addr*	*addr:* A-type address, or register (2) - (12). **Note:** If R, RC, or RU is coded, register (1) can also be specified.
,SP=*subpool nmbr*	*subpool nmbr:* symbol, decimal digit 0-127, or register (2) - (12). If R is specified above, register (0) may also be specified.
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

LC,LA = *length addr*
LU,LA = *length addr*
L,LA = *length addr*
VC
VU
V
EC,LV = *length value*
EU,LV = *length value*
E,LV = *length value*
RC,LV = *length value*
RC,SP = *subpool nmbr*
RU,LV = *length value*
RU,SP = *subpool nmbr*
R,LV = *length value*
R,SP = *subpool nmbr*
specifies the type of FREEMAIN request:

LC, LU, and L indicate conditional (LC) and unconditional (LU and L) list requests, and specify release of one or more areas of virtual storage. The length of each virtual storage area is indicated by the values in a list beginning at the address specified in the LA parameter. The address of each of the virtual storage areas must be provided in a corresponding list whose address is specified in the A parameter. All virtual storage areas must start on a doubleword boundary.

VC, VU, and V indicate conditional (VC) and unconditional (VU and V) variable requests, and specify release of single areas of virtual storage. The address and length of the virtual storage area are provided at the address specified in the A parameter.

EC, EU, and E indicate conditional (EC) and unconditional (EU and E) element requests, and specify release of single areas of virtual storage. The length of the single virtual storage area is indicated in the LV parameter. The address of the virtual storage area is provided at the address indicated in the A parameter.

RC, RU, and R indicate conditional (RC) and unconditional (RU and R) register requests, and specify release of single areas of virtual storage from the subpool indicated, or specify release of the entire subpool indicated. If the release is not for the entire subpool, the address of the virtual storage area is indicated in the A parameter. The length of the area is indicated in the LV parameter. The virtual storage area must start on a doubleword boundary.

Notes:

1. *A conditional request indicates that the task is not to be abnormally terminated if the virtual storage being freed is not allocated to the active task. However, A05-2 and A78-2 abends cannot be prevented. An unconditional request indicates that the task is to be abnormally terminated in this situation.*

2. *If the address of the area to be freed is greater than 16 Mb, you must use RC or RU.*

3. *Callers executing in either 24-bit or 31-bit addressing mode can use RC or RU to free storage located above 16 Mb.*

LA specifies the virtual storage address of one or more consecutive fullwords starting on a fullword boundary. One word is required for each virtual storage area to be released; the high-order bit in the last word must be set to 1 to indicate the end of the list. Each word must contain the required length in the low-order three bytes. The fullwords in this list must correspond with the fullwords in the associated list specified in the A parameter. The words should not reside in the area to be released. If this rule is violated and if the words are the last allocated items on a virtual page, the whole page is returned to storage and the FREEMAIN abends with an 0C4. The words must not overlap the virtual storage area specified in the A parameter.

LV specifies the length, in bytes, of the virtual storage area being released. The value should be a multiple of 8; if it is not, the control program uses the next high multiple of 8. If R is coded, LV=(0) may be designated; the high-order byte of register 0 must contain the subpool number, and the low-order three bytes must contain the length (in this case, the SP parameter is invalid).

,A=*addr*
specifies the virtual storage address of one or more consecutive fullwords, starting on a fullword boundary. The words should not reside within the area to be released. If this rule is violated and if the words are the last allocated items on a virtual page, the whole page is returned to storage and the FREEMAIN abends with an 0C4. If E, EC, EU, R, RC, or RU is designated, one word, which contains the address of the virtual storage area to be released, is required. If V, VC, or VU is coded, two words are required; the first word contains the address of the virtual storage area to be released, and the second word contains the length of the area. If L, LC, or LU is coded, one word is required for each virtual storage area to be released; each word contains the address of one virtual storage area. If R, RC, or RU is coded, any of the registers 1 through 12 can be designated, in which case the address of the virtual storage area, not the address of the fullword, must have previously been loaded into the register.

,SP=*subpool nmbr*
specifies the subpool number of the virtual area to be released. The subpool number can be between 0 and 127. The SP parameter is optional and if omitted, subpool 0 is assumed. If R is coded, SP=(0) can be designated, in which case the subpool number must be previously loaded into the low-order byte of register 0.

For subpool freemains, the SP parameter specifies the number of the subpool to be released and the valid range is 1 through 127. Subpool zero cannot be released. If R,SP=(0) is specified with no other parameters, the high-order byte of register 0 must contain the subpool number and the low-order 3 bytes must contain zero.

,RELATED=*value*
specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
GET1     GETMAIN     R,LV=4096,RELATED=(FREE1,
                     'GET STORAGE')
  .
  .
  .
FREE1    FREEMAIN    R,LV=4096,A=(1),
                     RELATED=(GET1'FREE STORAGE')
```

Note: Each of these macro instructions will fit on one line when coded, so there is no need for a continuation indicator.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Virtual storage was freed.
04	Not all virtual storage was freed.

Example 1

Operation: Free 400 bytes of storage from subpool 10, where the storage address is contained in register 1. If the storage was allocated to the task, register 15 will contain 0 on return; if the storage was not allocated to the task or was partially free, the status of the storage remains unchanged, and a 4 is returned in register 15.

```
FREEMAIN     RC,LV=400,A=(1),SP=10
```

Example 2

Operation: Free all of subpool 3 (if any) that belongs to the current task. A return will be made to the caller even if there is no subpool 3 for the current task.

```
FREEMAIN     RU,SP=3
```

Example 3

Operation: Free from subpool 5 three areas of lengths 200, 800, and 32 previously obtained by a list type GETMAIN which placed the addresses in AREADD. If any of these areas are not allocated to the task, the task will be abnormally terminated.

```
FREEMAIN     LU,LA=LNTHLIST,A=AREAADD,SP=5
  .
  .
  .
LNTHLIST     DC F'200',F'800',X'80',FL3'32'
AREAADD      DS 3F
```

FREEMAIN (List Form)

Use the list form of the FREEMAIN macro instruction to construct a nonexecutable control program parameter list.

The list form of the FREEMAIN macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede FREEMAIN.
FREEMAIN	
b	One or more blanks must follow FREEMAIN.

LC LU L VC VU V EC EU E	
,LA = *length addr* ,LV = *length value*	*length addr:* A-type address. *length value:* symbol or decimal digit. **Note:** LA may only be specified with LC, LU, or L above. **Note:** LV may only be specified with EC, EU, or E above.
,A = *addr*	*addr:* A-type address.
,SP = *subpool nmbr*	*subpool nmbr:* symbol or decimal digit 0-127.
,RELATED = *value*	*value:* any valid macro keyword specified.
,MF = L	

The parameters are explained under the standard form of the FREEMAIN macro instruction, with the following exception:

,MF = L
specifies the list form of the FREEMAIN macro instruction.

FREEMAIN (Execute Form)

A remote control program parameter list is used in, and can be modified by, the execute form of the FREEMAIN macro instruction. The parameter list can be generated by the list form of either a GETMAIN or a FREEMAIN.

The execute form the the FREEMAIN macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede FREEMAIN.
FREEMAIN	
b	One or more blanks must follow FREEMAIN.
LC LU L VC VU V EC EU E ,LA=*length addr* ,LV=*length value*	*length addr:* RX-type address or register (2) - (12). *length value:* symbol, decimal digit, or register (2) - (12). **Note:** LA may only be specified with LC, LU, or L above. **Note:** LV may only be specified with EC, EU, or E above.
,A=*addr*	*addr:* RX-type address, or register (2) - (12).
,SP=*subpool nmbr*	*subpool nmbr:* symbol, decimal digit 0-127, or register (2) - (12).
,RELATED=*value*	*value:* any valid macro keyword specified.
,MF=(E,*ctrl prog*)	*ctrl prog:* RX-type address, or register (1) or (2).

The parameters are explained under the standard form of the FREEMAIN macro instruction, with the following exception:

,MF=(E,*ctrl prog*)
specifies the execute form of the FREEMAIN macro instruction using a remote control program parameter list.

GETMAIN — Allocate Virtual Storage

The GETMAIN macro instruction allocates one or more areas of virtual storage to the active task. The virtual storage areas are allocated from the specified subpool in the virtual storage area assigned to the associated job step. The virtual storage areas each begin on a doubleword or page boundary and are not cleared to 0 when allocated. (The storage is set to zero for the initial allocation of a page.) The total of the lengths specified must not exceed the length available. For most subpools the storage will be released when the task assigned ownership terminates, or through the use of the FREEMAIN macro instructions.

The options R, LC, LU, VC, VU, EC, or EU can be used by callers in either 24-bit or 31-bit addressing mode. If one of these options is specified, storage area addresses and lengths will be treated as 24-bit addresses and values. The parameter list addresses and the pointers to the length and address lists in the parameter lists (if present) will be treated as 31-bit addresses if the caller's addressing mode is 31-bit; otherwise, they will be treated as 24-bit addresses.

The options RU, RC, VRU, and VRC can be used by callers in either 24-bit or 31-bit addressing mode. However, all values and addresses will be treated as 31-bit values and addresses.

The standard form of the GETMAIN macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede GETMAIN.
GETMAIN	
b	One or more blanks must follow GETMAIN.

LC,LA=*length addr,A=addr* LU,LA=*length addr,A=addr* VC,LA=*length addr,A=addr* VU,LA=*length addr,A=addr* EC,LV=*length value,A=addr* EU,LV=*length value,A=addr* RC,LV=*length value* RU,LV=*length value* R,LV=*length value*	*length addr:* A-type address, or register (2) - (12). *length value:* symbol, decimal digit, or register (2) - (12). If R is specified, register (0) may also be specified. *addr:* A-type address, or register (2) - (12). **Note:** RC, RU, VRC, or VRU must be used to allocate storage with addresses greater than 16Mb.
VRC,LV=(*maximum length value, minimum length value*) VRU,LV=(*maximum length value, minimum length value*	*maximum length value:* symbol, decimal, digit, or register (2) - (12). *minimum length value:* symbol, decimal, digit, or register (2) - (12).
,SP=*subpool nmbr*	*subpool nmbr:* symbol, decimal digit 0-127, or register (2) - (12). **Note:** Subpools are specified as follows: • LC,LU,VC,VU,EC,EU,RC,RU,VRC, and VRU use the SP parameter. • R with LV not equal to (0) uses the SP parameter. • R with LV=(0) must use register 0. The low-order three bytes of register 0 must contain the length of the subpool, and the high-order byte must contain the number of the subpool.
,BNDRY=DBLWD ,BNDRY=PAGE	**Default:** BNDRY=DBLWD **Note:** This parameter may not be specified with R above.
,LOC=BELOW ,LOC=(BELOW,ANY) ,LOC=(ANY) ,LOC=(ANY,ANY) ,LOC=RES ,LOC=(RES,ANY)	**Default:** LOC=RES **Note:** This parameter can only be used with RC, RU, VRC, or VRU. On all other forms, the default, LOC=BELOW is used.
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

LC,LA = *length addr,A = addr*
LU,LA = *length addr,A = addr*
VC,LA = *length addr,A = addr*
VU,LA = *length addr,A = addr*
EC,LV = *length value,A = addr*
EU,LV = *length value,A = addr*
RC,LV = *length value*
RU,LV = *length value*
R,LV = *length value*
VRC,LV = (*maximum length value, minimum length value*)
VRU,LV = (*maximum length value, minimum length value*)
specifies the type of GETMAIN request:

LC and LU indicate conditional (LC) and unconditional (LU) list requests and specify requests for one or more areas of virtual storage. The length of each virtual storage area is indicated by the values in a list beginning at the address specified in the LA parameter. The address of each of the virtual storage areas is returned in a list beginning at the address specified in the A parameter. No virtual storage is allocated unless all of the requests in the list can be satisfied.

VC and VU indicate conditional (VC) and unconditional (VU) variable requests and specify requests for single areas of virtual storage. The length of the single virtual storage area is between the two values at the address specified in the LA parameter. The address and actual length of the allocated virtual storage area are returned by the control program at the address indicated in the A parameter.

EC and EU indicate conditional (EC) and unconditional (EU) element requests, and specify requests for single areas of virtual storage. The length of the single virtual storage area is indicated in the LV = *length value* parameter. The address of the allocated virtual storage area is returned at the address indicated in the A parameter.

RC indicates a conditional register request, and R and RU indicate unconditional register requests. RC, RU, and R specify requests for single areas of virtual storage. The length of the single virtual area is indicated in the LV = *length value* parameter. The address of the allocated virtual storage area is returned in register 1. (R generates the SVC 10 calling sequence, whereas RU and RC generate the SVC 120 and associated parameter format.)

VRC and VRU indicate variable register conditional (VRC) and unconditional (VRU) requests for a single area of virtual storage. The length returned will be between the maximum and minimum lengths specified by the parameter LV = (*maximum length value, minimum length value*). The address of the allocated virtual storage is returned in register 1 and the length in register 0.

Notes:

1. *A conditional request indicates that the task is not to be abnormally terminated if virtual storage is not allocated to the active task. An unconditional request indicates that the task is to be abnormally terminated in this situation.*

2. *The LC, LU, VC, VU, EC, EU, and R forms of the GETMAIN macro instruction can only be used to obtain virtual storage with addresses below 16 Mb. The RC, RU, VRC, and VRU forms of the GETMAIN macro instruction can be used to obtain virtual storage when the addresses are above 16 Mb or when the addresses are below 16 Mb.*

LA specifies the virtual storage address of consecutive fullwords starting on a fullword boundary. Each fullword must contain the required length in the low-order three bytes, with the high-order byte set to 0. The lengths should be multiples of 8; if they are not, the control program uses the next higher multiple of 8. If VC or VU was coded, two words are required. The first word contains the minimum length required, the second word contains the maximum length. If LC or LU was coded, one word is required for each virtual storage area requested; the high-order bit of the last word must be set to 1 to indicate the end of the list. The list must not overlap the virtual storage area specified in the A parameter.

LV=*length value* specifies the length, in bytes, of the requested virtual storage. The number should be a multiple of 8; if it is not, the control program uses the next higher multiple of 8. If R is specified, LV=(0) may be coded; the low-order three bytes of register 0 must contain the length, and the high-order byte must contain the subpool number. LV=(*maximum length value*, *minimum length value*) specifies the maximum and minimum values of the length of the storage request.

A specifies the virtual storage address of consecutive fullwords, starting on a fullword boundary. The control program places the address of the virtual storage area allocated in one or more words. If E was coded, one word is required. If L was coded, one word is required for each entry in the LA list. If V was coded, two words are required. The first word contains the address of the virtual storage area, and the second word contains the length actually allocated. The list must not overlap the virtual storage area specified in the LA parameter.

,SP=*subpool nmbr*
specifies the number of the subpool from which the virtual storage area is to be allocated. The subpool number must be a valid subpool number between 0 and 127.

,BNDRY=DBLWD
,BNDRY=PAGE
specifies that alignment on a doubleword boundary (DBLWD) or alignment with the start of a virtual page on a 4K boundary (PAGE) is required for the start of a requested area. If one of the following subpools 226, 233-235, 239, 245, or 253-255 is requested, the BNDRY=PAGE keyword is ignored. Requests for storage from these subpools are assigned from a single page, unless the request is greater than a page.

,LOC=BELOW
,LOC=(BELOW,ANY)
,LOC=ANY
,LOC=(ANY,ANY)
,LOC=RES
,LOC=(RES,ANY)

specifies the location of virtual and real storage. When LOC is specified, real storage is allocated anywhere until the storage is fixed. After the storage is fixed, virtual and real storage are located in the following manner.

LOC=BELOW indicates that real and virtual storage are to be located below 16 Mb.

LOC=(BELOW,ANY) indicates that virtual storage is to be located below 16 Mb and real storage can be located anywhere.

LOC=ANY and LOC=(ANY,ANY) indicates that virtual and real storage can be located anywhere.

Note: The LOC parameter is not valid for fixed subpools. For fixed subpools the actual location of the virtual storage area depends on the subpool specified. If the subpool is supported (backed) above 16 megabytes, GETMAIN attempts to locate the virtual storage area above 16 megabytes. If this is not possible, GETMAIN locates the virtual storage below 16 megabytes. LSQA subpools will be backed anywhere regardless of the LOC parameter.

LOC=RES indicates that the location of virtual and real storage depends on the location of the caller. If the caller resides below 16 Mb, virtual and real storage are to be located below 16 Mb; if the caller resides above 16 Mb, virtual and real storage are to be located anywhere.

LOC=(RES,ANY) indicates that the location of virtual storage depends upon the location of the caller. If the caller resides below 16 Mb, virtual storage is to be located below 16 Mb; if the caller resides above 16 Mb, virtual storage can be located anywhere. In either case, real storage can be located anywhere.

,RELATED=*value*

specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
GET1     GETMAIN     R,LV=4096,RELATED=(FREE1,
                     'GET STORAGE')
   .
   .
   .
FREE1    FREEMAIN    R,LV=4096,A=(1),RELATED=(GET1,
                     'FREE STORAGE')
```

Note: Each of these macro instructions will fit on one line when coded, so there is no need for a continuation indicator.

When control is returned, for conditional type requests (LC, EC, VC, RC, and VRC) register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Virtual storage requested was allocated.
04	The request could not be satisfied because of insufficient virtual storage.

The contents of registers 0, 1, and 15 are not preserved when the GETMAIN macro instruction is issued.

Example 1

Operation: Obtain 400 bytes of storage from subpool 10. If the storage is available, the address will be returned in register 1 and register 15 will contain 0; if virtual storage is not available, register 15 will contain 4.

```
GETMAIN     RC,LV=400,SP=10
```

Example 2

Operation: Obtain 48 bytes of storage from default subpool 0. If the storage is available, the address will be stored in the word at AREAADDR; if the virtual storage is not available, the task will be abnormally terminated.

```
GETMAIN     EU,LV=48,A=AREAADDR
   .
   .
   .
AREAADDR    DS   F
```

Example 3

Operation: Obtain a maximum of 4096 or a minimum of 1024 bytes of virtual storage, with addresses above or below 16 Mb. Indicate that if the real storage is fixed, it should also be located above or below 16 Mb. If the storage is available, the address will be returned in register 1 and the length of the storage allocated will be returned in register 0; if the storage is not available, the task will be terminated.

```
GETMAIN VRU,LV=(4096,1024),LOC=ANY
```

GETMAIN (List Form)

Use the list form of the GETMAIN macro instruction to construct a control program parameter list.

The list form of the GETMAIN macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede GETMAIN.
GETMAIN	
b	One or more blanks must follow GETMAIN.

LC LU VC VU EC EU	
,LA=*length addr* ,LV=*length value*	*length addr:* A-type address. *length value:* symbol or decimal digit. **Note:** LA may only be specified with EC or EU above. **Note:** LV may only be specified with LC, LU, or VU above.
,A=*addr*	*addr:* A-type address.
,SP=*subpool nmbr*	*subpool nmbr:* symbol or decimal digit 0-127.
,BNDRY=DBLWD ,BNDRY=PAGE	**Default:** BNDRY=DBLWD
,RELATED=*value*	*value:* any valid macro keyword specified.
,MF=L	

The parameters are explained under the standard form of the GETMAIN macro instruction, with the following exception:

,MF=L
specifies the list form of the GETMAIN macro instruction.

GETMAIN (Execute Form)

A remote control program parameter list is used in, and can be modified by, the execute form of the GETMAIN macro instruction. The parameter list can be generated by the list form of either a GETMAIN or a FREEMAIN.

The execute form of the GETMAIN macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede GETMAIN.
GETMAIN	
b	One or more blanks must follow GETMAIN.

LC LU VC VU EC EU	
,LA=*length addr* ,LV=*length value*	*length addr:* RX-type address or register (2) - (12). *length value:* symbol, decimal digit, or register (2) - (12). **Note:** LA may only be specified with EC or EU above. **Note:** LV may only be specified with LC, LU, VC, or VU above.
,A=*addr*	*addr:* RX-type address, or register (2) - (12).
,SP=*subpool nmbr*	*subpool nmbr:* symbol, decimal digit 0-127, or register (2) - (12).
,BNDRY=DBLWD ,BNDRY=PAGE	**Default:** BNDRY=DBLWD
,RELATED=*value*	*value:* any valid macro keyword specified.
,MF=(E,*ctrl prog*)	*ctrl prog:* RX-type address, or register (1) or (2) - (12).

The parameters are explained under the standard form of the GETMAIN macro instruction, with the following exception:

,MF=(E,*ctrl prog*)
specifies the execute form of the GETMAIN macro instruction using a remote control program parameter list.

IDENTIFY — Add an Entry Name

The IDENTIFY macro instruction adds an entry name to a copy of a load module currently in virtual storage. The copy must be one of the following:

- A copy that satisfied the requirements of a LOAD macro instruction issued during the execution of the current task.
- The last load module given control, if control was passed to the load module using a LINK, ATTACH, or XCTL macro instruction.

The IDENTIFY macro instruction may not be issued by an asynchronous exit routine. Normally, the IDENTIFY macro assigns the identified entry point as reentrant. A user issuing this macro should be sure that his program is reenterable, otherwise, results are unpredictable.

An exception is the case of a non-authorized user identifying a module from an authorized library. In this case, the identified entry point is assigned the same attributes (reentrant, serially reusable, non-reusable, load only) as the main entry point. If the program is marked non-reusable, an ABEND 806 with a return code of four may result. The user should ensure that the program issuing this macro is reentrant or serially reusable if this exception applies.

IDENTIFY services sets the addressing mode of the entry name that was added equal to the addressing mode of the major entry name. The system assigns the major entry name according to how the load module was constructed.

- If the load module was constructed using the linkage editor (and brought into virtual storage via program fetch or virtual fetch), the major entry name is the name of the load module in the partitioned data set directory (not an alias to that member).
- If the load module was brought into storage by the loader, the major entry name is either the name that the user provided as input to the loader or the name that the loader used as a default. See the NAME= parameter in the LOADER section of *Linkage Editor and Loader* for information about how to specify this name.

Note: You can override the addressing mode of the entry name by using the AMODE parameter in the PARM field of the EXEC JCL statement.

If an authorized caller creates an entry name for a module in the pageable link pack area, IDENTIFY services places an entry for the alias on the active link pack area queue. If an unauthorized caller creates an entry name for a module in the pageable link pack area, IDENTIFY services places an entry for the alias on the task's job pack queue.

The IDENTIFY macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede IDENTIFY.
IDENTIFY	
b	One or more blanks must follow IDENTIFY.
EP=*entry name*	*entry name:* symbol
EPLOC=*entry name addr*	*entry name addr:* RX-type address, or register (0) or (2) - (12).
,ENTRY=*entry addr added*	*entry addr added:* RX-type address, or register (1) or (2) - (12).

The parameters are explained as follows:

EP = *entry name*
EPLOC = *entry name addr*
specifies the entry name or address of the entry name. The name does not have to correspond to any symbol or name in the load module, and must not correspond to any name, alias, or added entry name for a load module in the link pack area queue, or the job pack area of the job step. If EPLOC is coded, the name must be padded to eight bytes, if necessary.

,ENTRY = *entry addr added*
specifies the virtual storage address of the entry point being added.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion of requested function.
04	Entry name and address already exist.
08	Entry name duplicates the name of a load module currently in virtual storage; entry address was not added.
0C	Entry address is not within an eligible load module; entry address was not added.
10	Request issued by an asynchronous exit routine; entry address was not added.
14	LINK, LOAD, XCTL, ATTACH, or IDENTIFY request was previously issued using the same entry name but a different address; current request was ignored.
18	Parameter list is invalid or is not on a word boundary.
1C	Extent list length is not positive or a multiple of 8, or extent address is not on a double word boundary, is not addressable, or is not in caller's region.
24	Unexpected system error.
28	EPLOC address is fetch protected.

Example 1

Operation: Add an entry name (PGMTAL2A) to a load module in virtual storage. Register 3 contains the entry point address.

```
IDENTIFY      EP=PGMTAL2A,ENTRY=(R3)
```

LINK — Pass Control to a Program in Another Load Module

The LINK macro instruction passes control to a specified entry name in another load module; the entry name must be a member name or an alias in a directory of a partitioned data set (PDS) or must have been specified in an IDENTIFY macro instruction. The load module containing the program is brought into virtual storage if a usable copy is not available.

LINK processing handles the setting of the addressing mode when passing control. The called program is given control in the addressing mode indicated in its PDS directory entry. On entry to the called program, the high-order bit, bit 0, of register 14 is set to indicate the addressing mode of the issuer of the LINK macro. If bit 0 is 0, the issuer is executing in 24-bit addressing mode; if bit 0 is 1, the issuer is executing in 31-bit addressing mode. This makes it possible to return control to the calling program in the addressing mode in which it was executing.

The problem program optionally can provide a parameter list to be passed to the called program. If the called program terminates abnormally, or if the specified entry point cannot be located, the task is abnormally terminated.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction.

The standard form of the LINK macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede LINK.
LINK	
b	One or more blanks must follow LINK.

EP=*entry name* EPLOC=*entry name addr* DE=*list entry addr*	*entry name:* symbol *entry name addr:* A-type address, or register (2) - (12). *list entry addr:* A-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* A-type address, or register (2) - (12).
,PARAM=(*addr*) ,PARAM=(*addr*),VL=1	*addr:* A-type address, or register (2) - (12). **Note:** *addr* is one or more addresses, separated by commas. For example, (*addr,addr,addr*)
,ID=*id nmbr*	*id nmbr:* symbol or decimal digit, with a maximum value of 4095.
,ERRET=*err rtn addr*	*err rtn addr:* A-type address, or register (2) - (12).
,LSEARCH=NO ,LSEARCH=YES	**Default:** No

The parameters are explained as follows:

EP=*entry name*
EPLOC=*entry name addr*
DE=*list entry addr*

specifies the entry name, the address of the entry name, or the address of the name field in a 60-byte list entry for the entry name that was constructed using the BLDL macro instruction. If EPLOC is coded, the name must be padded to eight bytes, if necessary.

If you issue the LINK macro instruction and the DE parameter specifies an entry in an authorized library, the program-supplied DE information is ignored for integrity reasons.

Instead, contents management uses the BLDL macro instruction to construct a new list entry containing the DE information for the LINK.

The DE information supplied by an unauthorized program will also be ignored if the LINK macro instruction is requesting access to a program or library that is controlled by the System Authorization Facility.

Note: When you use the DE parameter with the LINK macro, DE specifies the address of a list that was created by a BLDL macro. BLDL and LINK must be issued from the same task; otherwise, the system might terminate the program with an abend code of 106 and a return code of 15. Therefore, do not issue an ATTACH or a DETACH macro between issuances of the BLDL and LINK macros.

,DCB=*dcb addr*
specifies the address of the opened data control block for the partitioned data set containing the entry name described above. This parameter must indicate the same DCB used in the BLDL mentioned above.

If the DCB parameter is omitted or if DCB=0 is specified when the LINK macro instruction is issued by the *job step task,* the data sets referred to by either the STEPLIB or JOBLIB DD statement are first searched for the entry point name. If the entry point name is not found, the link library is searched.

If the DCB parameter is omitted or if DCB=0 is specified when the LINK macro instruction is issued by a *subtask*, the data sets associated with one or more data control blocks referred to by the TASKLIB operand of previous ATTACH macro instructions in the subtasking chain are first searched for the entry point name. If the entry point name is not found, the search is continued as if LINK had been issued by the job step task.

Note: DCB must reside in 24-bit addressable storage.

,PARAM=(*addr***)**
,PARAM=(*addr***),VL=1**
specifies address(es) to be passed to the called program. Each address is expanded inline to a fullword on a fullword boundary, in the order designated. Register 1 contains the address of the first parameter when the program is given control. (If this parameter is not coded, register 1 is not altered unless the execute form of the LINK macro instruction is coded.)

VL=1 should be designated only if the called program can be passed a variable number of parameters. VL=1 causes the high-order bit of the last address parameter to be set to 1; the bit can be checked to find the end of the list.

,ID=*id nmbr*
specifies an identifier useful for debugging purposes only. The last fullword of the macro expansion is a NOP instruction containing the identifier value in bytes 3 and 4.

,ERRET=*err rtn addr*
specifies the address of a routine to receive control when an error condition that would cause an abnormal termination of the task is detected. Register 1 contains the abend code that would have resulted had the task abended. The routine does not receive control when input parameter errors are detected.

,LSEARCH=NO
,LSEARCH=,YES
specifies whether (YES) or not (NO) the search is to be limited to the job pack area and the first library in the normal search sequence.

Example 1

Operation: Pass control to a specified entry name (PGMLKRUS) in another load module. Let the system find the module from available libraries.

```
LINK     EP=PGMLKRUS
```

Example 2

Operation: Pass control to a specified entry name (PGMA) in another load module, specifying (in registers 4, 6, 8) three addresses to be passed to the called program.

```
LINK EP=PGMA,PARAM=((4),(6),(8))
```

LINK (List Form)

Two parameter lists are used in a LINK macro instruction: a control program parameter list and problem program parameter list. Only the control program parameter list can be constructed in the list form of LINK. Address parameters to be passed in a parameter list to the problem program can be provided using the list form of CALL. This parameter list can be referred to in the execute form of LINK.

The list form of the LINK macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede LINK.
LINK	
b	One or more blanks must follow LINK.

EP=*entry name* EPLOC=*entry name addr* DE=*list entry addr*	*entry name:* symbol *entry name addr:* A-type address. *list entry addr:* A-type address.
,DCB=*dcb addr*	*dcb addr:* A-type address.
,ERRET=*err rtn addr*	*err rtn addr:* A-type address.
,LSEARCH=NO ,LSEARCH=YES	**Default:** No
,SF=L	

The parameters are explained under the standard form of the LINK macro instruction, with the following exception:

,SF=L
specifies the list form of the LINK macro instruction.

Notes:

1. *Coding the LSEARCH parameter causes a parameter list to be created that is different from the list created when LSEARCH is omitted. If you code LSEARCH in either the list or execute form of the macro instruction, you must code it in both forms.*

2. *If ERRET is coded in the list form and not specified in the execute form, the error routine specified in the list form will be retained and used in the execute form of the macro instruction. If ERRET is specified in both the list and the execute form, the error routine specified in the execute form of the macro instruction will be used.*

LINK (Execute Form)

Two parameter lists are used in a LINK macro instruction: a control program parameter list and an optional problem program parameter list. Either or both of these lists can be remote and can be referred to and modified by the execute form of LINK. If only one of the parameter lists is remote, parameters that require use of the other parameter list cause that list to be constructed inline as part of the macro expansion.

The execute form of the LINK macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede LINK.
LINK	
b	One or more blanks must follow LINK.

EP=*entry name* EPLOC=*entry name addr* DE=*list entry addr*	*entry name:* symbol. *entry name addr:* RX-type address or register (2) - (12). *list entry addr:* RX-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* RX-type address, or register (2) - (12).
,PARAM=(*addr*) ,PARAM=(*addr*),VL=1	*addr:* RX-type address, or register (2) - (12). **Note:** *addr* is one or more addresses, separated by commas. For example, (*addr,addr,addr*)
,ID=*id nmbr*	*id nmbr:* symbol or decimal digit, with a maximum value of 4095.
,ERRET=*err rtn addr*	*err rtn addr:* RX-type address or register (2) - (12).
,LSEARCH=NO ,LSEARCH=YES	**Default:** No
,MF=(E,*prob addr*) ,SF=(E,*ctrl addr*) ,MF=(E,*prob addr*),SF=(E,*ctrl addr*)	*prob addr:* RX-type address, or register (1) or (2) - (12). *ctrl addr:* RX-type address, or register (2) - (12) or (15).

The parameters are explained under the standard form of the LINK macro instruction, with the following exceptions:

,MF=(E,*prob addr*)
,SF=(E,*ctrl addr*)
,MF=(E,*prob addr*),SF=(E,*ctrl addr*)
specifies the execute form of the LINK macro instruction. This form uses a remote problem program parameter list, a remote control program parameter list, or both.

Notes:

1. *Coding the LSEARCH parameter causes a parameter list to be created that is different from the list created when LSEARCH is omitted. If you code LSEARCH in either the list or the execute form of the macro instruction, you must code it in both forms.*

2. *If ERRET is coded in the list form and not specified in the execute form, the error routine specified in the list form will be retained and used in the execute form of the macro instruction. If ERRET is specified in both the list and the execute form, the error routine specified in the execute form of the macro instruction will be used.*

LOAD — Bring a Load Module into Virtual Storage

The LOAD macro instruction brings the load module containing the specified entry name into virtual storage, if a usable copy is not available in virtual storage.

LOAD services places the load module in storage above or below the 16 megabyte line depending on the module's RMODE, which is specified in the partitioned data set's directory entry for the module.

The responsibility count for the load module is increased by one. On output, the high-order byte of register 1 contains the authorization code of the loaded module and the low three bytes contain the module's length in doublewords. Control is *not* passed to the load module; instead, the virtual storage address of the designated entry point is returned in register 0. The load module remains in virtual storage until the responsibility count is reduced to 0 through task terminations or until the effects of all outstanding LOAD requests for the module have been canceled (using the DELETE macro instruction), *and* there is no other requirement for the module.

LOAD services sets the high order bit of the entry point address in register 0 to indicate the module's AMODE, which is obtained from the partitioned data set's directory entry for the module. If the module's AMODE is 31, it sets the indicator to 1; if the module's AMODE is 24, it sets the indicator to 0; and if the module's AMODE is ANY, it sets the indicator to correspond to the caller's AMODE.

The entry name in the load module must be a member name or an alias in a directory of a partitioned data set or must have been specified in the IDENTIFY macro instruction. If the entry name was previously specified in an IDENTIFY macro instruction, no attempt is made to bring in an additional copy of the module. If the specified entry name cannot be located, the task is abnormally terminated.

The LOAD macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede LOAD.
LOAD	
b	One or more blanks must follow LOAD.

EP=*entry name*	*entry name:* symbol.
EPLOC=*entry name addr*	*entry name addr:* If LSEARCH or LOADPT is specified, A-type address or register (2) - (12); otherwise, RX-type address or register (0) or (2) - (12).
DE=*list entry addr*	*list entry addr:* If LSEARCH or LOADPT is specified, A-type address or register (2) - (12); otherwise, RX-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* If LSEARCH or LOADPT is specified, A-type address or register (2) - (12); otherwise, RX-type address, or register (1) or (2) - (12).
,ERRET=*err rtn addr*	*err rtn addr:* RX-type address or register (2) - (12).
,LSEARCH=NO ,LSEARCH=YES	**Default:** No
,LOADPT=*addr*	*addr:* A-type address or register (2) - (12).
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained as follows:

EP = *entry name*
EPLOC = *entry name addr*
DE = *list entry addr*
specifies the entry name, the address of the name, or the address of the name field in a 60-byte list entry for the entry name that was constructed using the BLDL macro instruction. If EPLOC is coded, the name must be padded to eight bytes, if necessary.

If you issue the LOAD macro instruction and the DE parameter specifies an entry in an authorized library, the program-supplied DE information is ignored for integrity reasons. Instead, contents management uses the BLDL macro instruction to construct a new list entry containing the DE information for the LOAD. The DE information supplied by an unauthorized program will also be ignored if the LOAD macro instruction is requesting access to a program or library that is controlled by the System Authorization Facility.

Note: When you use the DE parameter with the LOAD macro, DE specifies the address of a list that was created by a BLDL macro. BLDL and LOAD must be issued from the same task; otherwise, the system might terminate the program with an abend code of 106 and a return code of 15. Therefore, do not issue an ATTACH or a DETACH macro between issuances of the BLDL and LOAD macros.

,DCB = *dcb addr*
specifies the address of the opened data control block for the partitioned data set containing the entry name described above. This parameter must indicate the same DCB used in the BLDL mentioned above.

If the DCB parameter is omitted or if DCB = 0 is specified when the LOAD macro instruction is issued by the *job step task*, the data sets referred to by either the STEPLIB or JOBLIB DD statement are first searched for the entry name. If the entry name is not found, the link library is searched.

If the DCB parameter is omitted or if DCB = 0 is specified when the LOAD macro instruction is issued by a *subtask*, the data sets associated with one or more data control blocks referred to by the TASKLIB operand of previous ATTACH macro instructions in the subtasking chain are first searched for the entry name. If the entry name is not found, the search is continued as if the LOAD had been issued by the job step task.

Note: DCB must reside in 24-bit addressable storage.

,ERRET = *err rtn addr*
specifies the address of a routine to receive control when an error condition that would cause an abnormal termination of the task is detected. Register 1 contains the abend code that would have resulted had the task abended, and register 15 contains the reason code that is associated with the abend. The routine does not receive control when input parameter errors are detected.

,LSEARCH = NO
,LSEARCH = YES
specifies whether (YES) or not (NO) the search is to be limited to the job pack area and the first library in the normal search sequence.

,LOADPT = *addr*

specifies that the starting address at which the module was loaded is to be returned to the caller at the indicated address.

,RELATED = *value*

specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
LOAD1    LOAD        EP=APGIOHK1,RELATED=(DEL1,
                     'LOAD APGIOHK1')
   .
   .
   .
DEL1     DELETE      EP=APGIOHK1,RELATED=(LOAD1,
                     'DELETE APGIOHK1')
```

Note: Each of these macro instructions will fit on one line when coded, so there is no need for a continuation indicator.

Example 1

Operation: Bring a load module containing a specified entry name (PGMLKRUS) into virtual storage. Let the system find the module from available libraries.

```
LOAD      EP=PGMLKRUS
```

Example 2

Operation: Bring a load module containing the entry name EPNAME into virtual storage. Indicate that register 7 contains the address of the DCB associated with the partitioned data set that contains this load module. Return the load address of the requested module in the location pointed to by register 8. If an error occurs during this processing, transfer control to the error routine located at ERRADDR.

```
LOAD EP=EPNAME,DCB=(7),LOADPT=(8),ERRET=ERRADDR
```

LOAD (List Form)

The list form of the LOAD macro instruction builds a non-executable problem program parameter list that can be referred to or modified by the execute form of the LOAD macro instruction.

The list form of the LOAD macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede LOAD.
LOAD	
b	One or more blanks must follow LOAD.
EP=*entry name*	*entry name:* symbol.
EPLOC=*entry name addr*	*entry name addr:* A-type address.
DE=*list entry addr*	*list entry addr:* A-type address.
,DCB=*dcb addr*	*dcb addr:* A-type address.
,LSEARCH=NO ,LSEARCH=YES	**Default:** No
,LOADPT=*addr*	*addr:* A-type address.
,RELATED=*value*	*value:* any valid macro keyword specification.
,SF=L	

The parameters are explained under the standard form of the LOAD macro instruction with the following exception:

,SF=L
specifies the list form of the LOAD macro instruction.

LOAD (Execute Form)

The execute form of the LOAD macro instruction can refer to and modify the parameter list constructed by the list form of the macro instruction.

The execute form of the LOAD macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede LOAD.
LOAD	
b	One or more blanks must follow LOAD.
EP=*entry name*	*entry name:* symbol.
EPLOC=*entry name addr*	*entry name addr:* RX-type address, or register (2) - (12).
DE=*list entry addr*	*list entry addr:* RX-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* RX-type address, or register (2) - (12).
,ERRET=*err rtn addr*	*err rtn addr:* RX-type address, or register (2) - (12).
,LSEARCH=NO ,LSEARCH=YES	**Default:** No
,LOADPT=*addr*	*addr:* RX-type address or register (2) - (12).
,RELATED=*value*	*value:* any valid macro keyword specification.
,SF=(E,*list addr*)	*list addr:* RX-type address or register (2) - (12) or (15).

The parameters are explained under the standard form of the LOAD macro instruction with the following exception:

,SF=(E,*list addr*)
specifies the execute form of the LOAD macro instruction.

PGLOAD — Load Virtual Storage Areas into Real Storage

The PGLOAD macro instruction loads specified virtual storage areas into real storage in anticipation of future needs. That is, PGLOAD is essentially a page-ahead function. The PGLOAD macro instruction performs this function for virtual addresses below 16 Mb; the LOAD option of the PGSER macro instruction performs the same function for virtual addresses either above or below 16 Mb. Note, however, that a page that has been loaded via PGLOAD is eligible for page-out selection in the same manner as a page that has been demand-paged into real storage.

The misuse of this function can have adverse effects on system performance. Causing unnecessary pages to be brought into real storage will force more useful pages to be displaced and, consequently, cause unnecessary paging activity. Proper use of this function, however, will tend to decrease system overhead resulting from page faults.

The standard form of the PGLOAD macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede PGLOAD.
PGLOAD	
b	One or more blanks must follow PGLOAD.
R	
,A=*start addr*	*start addr:* A-type address, or register (1) or (2) - (12).
,ECB=*ecb addr*	*ecb addr:* A-type address, or register (0) or (2) - (12).
,EA=*end addr*	*end addr:* A-type address, or register (2) - (12) or (15). **Default:** *start addr* + 1
,RELEASE=N ,RELEASE=Y	**Default:** RELEASE=N **Note:** RELEASE=Y may only be specified with EA above.

The parameters are explained as follows:

R
specifies that no parameter list is being supplied with this request.

,A=*start addr*
specifies the start address of the virtual area to be loaded.

,ECB=*ecb addr*
specifies the address of an ECB that is used to signal event completion.

,EA=*end addr*
specifies the end address + 1 of the virtual area to be loaded.

,RELEASE=N
,RELEASE=Y
specifies that the contents of the virtual area is to remain intact (N) or be released (Y).

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Operation completed normally; ECB posted complete.
08	Operation proceeding; ECB will be posted when all page-ins are complete.

If control is not returned, an ABEND is issued with the following reason codes in register 15:

Hexadecimal Code	Meaning
10	Virtual subarea list entry or ECB address invalid. No ECB is posted.

If the ECB parameter is coded, the ECB is unchanged if the request was initiated but not complete (return code 8), or if an ABEND was issued with return code 10. Otherwise, the ECB is posted complete with code

0 - Operation completed successfully.

If the return code issued is 8, the ECB is posted asynchronously when paging I/O has completed, with code

0 - Operation completed successfully.

Example 1

Operation: Page-in a single byte of virtual storage, causing the entire 4096-byte page containing that byte to be paged into real storage.

```
PGLOAD   R,A=(R3)
```

Example 2

Operation: Page-in the virtual storage lying in the range addressed by registers 3 and 4, and notify the requestor via posting of the ECB when the page-ins are complete.

```
PGLOAD   R,A=(R3),EA=(R4),ECB=(R5)
```

Example 3

Operation: Discard the contents of the virtual pages totally encompassed by STARTAD and ENDAD before new real storage frames are assigned.

```
PGLOAD   R,A=STANDARD,EA=ENDAD,RELEASE=Y
```

PGLOAD (List Form)

The list form of the PGLOAD macro instruction uses a virtual subarea list.

The list form of the PGLOAD macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede PGLOAD.
PGLOAD	
b	One or more blanks must follow PGLOAD.
L	
,LA=*list addr*	*list addr:* A-type address, or register (1) or (2) - (12).
,ECB=*ecb addr*	*ecb addr:* A-type address, or register (0) - (2) or (15).
,RELEASE=N ,RELEASE=Y	**Default:** RELEASE=N

The parameters are explained under the standard form of the PGLOAD macro instruction, with the following exceptions:

L
specifies that a parameter list is being supplied with this request.

,LA=*list addr*
specifies the address of the first entry of a virtual subarea list.

PGOUT — Page Out Virtual Storage Areas from Real Storage

The PGOUT macro instruction initiates page-out operations for specified virtual storage areas that are in real storage. The PGOUT macro instruction performs this function for virtual addresses below 16 Mb; the OUT option of the PGSER macro instruction performs the same function for virtual addresses either above or below 16 Mb. The PGOUT function is complementary to the PGLOAD function. You have the option of specifying that the virtual pages to be paged out either remain valid in real storage or be marked invalid and the real frames assigned to them be made available for reuse. The use of this option will not prevent page faults from occurring on the specified storage.

The misuse of this function, like the misuse of the PGLOAD function, can have adverse effects on system performance. On the other hand, proper use of this function will tend to clean out of real storage those pages no longer needed for program execution or not required for some period in the future.

The standard form of the PGOUT macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede PGOUT.
PGOUT	
b	One or more blanks must follow PGOUT.

R	
,A=*start addr*	*start addr:* A-type address, or register (1) or (2) - (12).
,EA=*end addr*	*end addr:* A-type address, or register (2) - (12) or (15).
,KEEPREL=N ,KEEPREL=Y	**Default:** KEEPREL=N

The parameters are explained as follows:

R
specifies that no parameter list is being supplied with this request.

,A=*start addr*
specifies the start address of the virtual area to be paged out.

,EA=*end addr*
specifies the end address + 1 of the virtual area to be paged out.

,KEEPREL=N
,KEEPREL=Y
specifies that the virtual pages will be marked invalid and the real storage frames freed for reuse (N) or that the virtual pages will not be invalidated (Y).

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Operation completed normally; paging I/O proceeding asynchronously.
0C	One or more pages specified to be paged out were not paged out. Either the pages were in the nucleus in unusable real frames, in SQA or LSQA, in V=R area allocated region, were page fixed, or the system resources necessary to perform the page out operations were momentarily unavailable. Paging I/O is proceeding normally for all other pages.
10	Operation abnormally terminated. Virtual subarea list entry invalid.

Example 1

Operation: Page-out the area of real storage totally encompassed by the start and end virtual boundaries specified.

```
PGOUT    R,A=(R3),EA=(R4)
```

Example 2

Operation: Create an auxiliary storage copy of a virtual area before continuing to use the area. The area will remain in real storage after the page-outs complete.

```
PGOUT    R,A=(R3),EA=(R4),KEEPREL=Y
```

PGOUT (List Form)

The list form of the PGOUT macro instruction uses a virtual subarea list.

The list form of the PGOUT macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede PGOUT.
PGOUT	
b	One or more blanks must follow PGOUT.
L	
,LA=*list addr*	*list addr:* A-type address, or register (1) or (2) - (12).
,KEEPREL=N ,KEEPREL=Y	**Default:** KEEPREL=N

The parameters are explained under the standard form of the PGOUT macro instruction, with the following exceptions:

L
specifies that a parameter list is being supplied with this request.

,LA=*list addr*
specifies the address of the first entry of a virtual subarea list (VSL). See the topic "Virtual Subarea List (VSL)" in Part I for a description of the VSL.

PGRLSE — Release Virtual Storage Contents

The PGRLSE macro instruction releases to the system all real storage and auxiliary storage associated with specified pageable virtual storage areas. The PGRLSE macro instruction performs this function for virtual addresses below 16 Mb; the RELEASE option of the PGSER macro instruction performs the same function for virtual addresses either above or below 16 Mb. Use PGRLSE when a large area (one or more complete pages) of virtual storage within your program no longer has significant contents.

Functionally, PGRLSE is equivalent to a FREEMAIN macro instruction followed by a GETMAIN macro instruction. That is, the virtual space is maintained, but the data is discarded. When the page(s) being released is next referred to, that page is cleared to zeros. Thus, you can help reduce system overhead by releasing virtual storage when you no longer need it.

Proper use of this function can increase the amount of storage available to the system and prevent needless paging I/O activity. Usage of PGRLSE may improve operating efficiency when the using program can discard the contents of a large virtual storage area and reuse the virtual storage pages; paging operations may be eliminated for those virtual storage pages when they are reused.

The standard form of the PGRLSE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede PGRLSE.
PGRLSE	
b	One or more blanks must follow PGRLSE.
LA=*low addr*	*low addr:* A-type address, or register (0) or (2) - (12).
,HA=*high addr*	*high addr:* A-type address, or register (1) or (2) - (12).

The parameters are explained as follows:

LA=*low addr*
specifies the address of the lower boundary of the area to be released.

,HA=*high addr*
specifies the address of the upper boundary + 1 of the area to be released.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion.
04	Execution failed. The area specified, or a portion of the area, is protected from the requesting program. Any valid portion of the area preceding the protected area is released.

Example 1

Operation: Release the contents of the pages included within the specified areas. Only those pages fully encompassed will be nullified.

```
PGRLSE    LA=(R4),HA=(R5)
```

Example 2

Operation: Perform the operation in Example 1, but use A-type addresses.

```
PGRLSE    LA=LOWADDR,HA=HIGHADDR
```

PGRLSE (List Form)

The list form of the PGRLSE macro instruction constructs a control program parameter list.

The list form of the PGRLSE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede PGRLSE.
PGRLSE	
ƀ	One or more blanks must follow PGRLSE.
LA=*low addr*,	*low addr:* A-type address.
HA=*high addr*,	*high addr:* A-type address.
MF=L	

The parameters are explained under the standard form of the PGRLSE macro instruction, with the following exception:

MF=L
specifies the list form of the PGRLSE macro instruction.

PGRLSE (Execute Form)

A remote control program parameter list is referred to, and can be modified by, the execute form of the PGRLSE macro instruction.

The execute form of the PGRLSE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede PGRLSE.
PGRLSE	
b	One or more blanks must follow PGRLSE.
LA=*low addr*,	*low addr:* A-type address, or register (0) or (2) - (12).
HA=*high addr*,	*high addr:* A-type address, or register (1) or (2) - (12).
MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (2) - (12).

The parameters are explained under the standard form of the PGRLSE macro instruction, with the following exception:

MF=(E,*ctrl addr*)
specifies the execute form of the PGRLSE macro instruction using a remote control program parameter list.

PGSER — Page Services

The PGSER macro instruction performs the same paging services as PGLOAD, PGOUT, and PGRLSE. PGSER performs these services for addresses either above or below 16 Mb.

The services are:

- Page load equivalent to PGLOAD
- Page out equivalent to PGOUT
- Page release equivalent to PGRLSE

If the common area is being released and the caller is not in key zero, the caller's key must match the key of the storage.

Regardless of the addressing mode, all addresses passed in registers are used as 31-bit addresses. All RX-type addresses are assumed to be in the addressing mode of the caller. The standard form of the PGSER macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede PGSER.
PGSER	
b	One or more blanks must follow PGSER.

R L	
,LOAD ,OUT ,RELEASE	
,LA=*list addr*	*list addr:* RX-type address or register (1), (2) - (12). **Note:** This parameter is valid only with L.
,A=*start addr*	*start addr:* RX-type address or register (1), (2) - (12). **Note:** This parameter is valid only with R.
,EA=*end addr*	**Default:** EA=*start addr* *end addr:* RX-type address or register (15), (2) - (12). **Note:** This parameter is valid only with R.
,ECB=*ecb addr*	**Default:** If LOAD is specified, ECB=0. *ecb addr:* RX-type address or register (0) or (2) - (12). **Note:** This parameter is optional if LOAD is specified and is invalid for OUT and RELEASE.
,RELEASE=Y ,RELEASE=N	**Default:** RELEASE=N **Note:** This parameter may be specified only if LOAD is specified.
,KEEPREL=Y ,KEEPREL=N	**Default:** KEEPREL=N **Note:** This parameter may be specified only if OUT is specified.
,RELATED=*value*	*value:* any valid macro keyword specification.

R
L

specifies the manner in which the input is supplied. If R is specified, the user supplies the starting and ending addresses of the virtual area for which the service needs to be performed. Before processing the request, page services puts these addresses in registers 1 and 15, respectively. If L is specified, the user supplies the address of the page services list (PSL), which specifies the virtual area for which the service is to be performed. Before processing the request, page services puts the address of the PSL in register 1. See the topic "Page Service List (PSL)" in Part I for a description of the PSL.

,LOAD
,OUT
,RELEASE

indicates the function to be performed.

LOAD specifies that a page-in operation is to be initiated for the virtual storage area specified, in anticipation of future needs.

OUT specifies that a page-out operation is to be initiated for the virtual storage area specified.

RELEASE specifies that all real and auxiliary storage, associated with the virtual storage area specified, is to be released.

,LA=*list addr*

specifies the address of the page services list (PSL) for L requests.

,A=*start addr*

specifies the address of the start of the virtual area for R requests.

,EA=*end addr*

specifies the address of the end of the virtual area for R requests.

Note: PGLOAD, PGOUT, and PGRLSE use end + 1 as the value for the end address.

,ECB=*ecb addr*

specifies the address of the ECB that is used to signal event completion for a LOAD request.

If an ECB is supplied, the caller must check the return code because the ECB will not be posted if the return code is zero. If an ECB is not supplied, it is not necessary to check the return code because control returns to the caller only if the request was successfully completed; if unsuccessful, page services abnormally terminates the caller.

Page services verifies that the ECB address is in an area allocated via GETMAIN and that the ECB is in the caller's protect key. Before posting the ECB, page services again verifies that the ECB is located in an allocated area and that the ECB is in the caller's protect key. (This is to check that the allocated area has not been freed via FREEMAIN and the protect key has not been changed.) It is the user's responsibility to ensure that the page containing the ECB is not freed and that the key is not altered. If either test fails, page services does not post the ECB.

,RELEASE=Y
,RELEASE=N
specifies that all the real and auxiliary storage associated with the virtual storage areas is to be released to the system (Y) or that all the real and auxiliary storage associated with the virtual storage areas is not to be released to the system (N).

,KEEPREL=Y
,KEEPREL=N
specifies that the virtual pages should be validated again after the page-out completes (Y); or that the virtual pages will be marked invalid and the real storage frames freed for reuse (N).

,RELATED=*value*
provides information to document the macro by relating the service performed to some corresponding function or service. The format can be any valid coding value that the user chooses.

On return the register contents are 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	This register contains the return code.

The return codes, given in register 15, along with the option used and the meaning follow:

Option	Code	Meaning
LOAD	0	The operation completed normally and the ECB will not be posted. If no ECB is supplied, the operation is completed or proceeding.
LOAD	8	The operation is proceeding. The ECB, if applicable and available, will be posted with X'00' when all page-ins are complete.
OUT	0	The operation completed normally.
OUT	C	One or more pages specified to be paged-out was not paged out. The page service is proceeding for the other pages
RELEASE	0	The operation completed normally.

If an error is found in one of the parameters, the requestor is abnormally terminated with a system abend code of X'18A' and one of the following hexadecimal reason codes is provided in register 15:

Hexadecimal Code	Meaning
4	A page-release operation abnormally terminated because either a page release was attempted for permanently backed storage or a non-system key caller attempted to release storage in a different key.
10	A page-load operation abnormally terminated because the PSL or ECB address was invalid. (An ECB can be specified for a LOAD request.)

Callers not authorized to use a specific service are abnormally terminated with a system abend code of X'28A'.

Example 1

Operation: Perform the page-load function for the 4096-byte virtual area starting at BUFFER. No ECB is supplied.

```
PGSER R,LOAD,A=BUFFER,EA=BUFFER+4095,ECB=0
```

Example 2

Operation: Release the virtual area specified in the PSL located at LOADWORD.

```
PGSER L,RELEASE,LA=LOADWORD
```

POST — Signal Event Completion

The POST macro instruction sets the specified ECB (event control block) to indicate the occurrence of an event. If this event satisfies the requirements of an outstanding WAIT or EVENTS macro instruction, the waiting task is taken out of the wait state and dispatched according to its priority. POST sets the bits in the specified ECB as follows:

Bit 0 to 0 (wait bit)
Bit 1 to 1 (complete bit)
Bits 2 through 31 to the specified completion code

Note: After the bits in the ECB are set, the ECB is considered posted, and the awaited event can be recognized as having occurred by programs running in the system. If a program issues another POST against an ECB that is already posted, the other POST has no effect.

The POST macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede POST.
POST	
ƀ	One or more blanks must follow POST.
ecb addr	*ecb addr:* RX-type address, or register (1) or (2) - (12).
,comp code	*comp code:* symbol, decimal digit, or register (0) or (2) - (12). **Range of values:** 0 to 2^{30}-1 **Default:** 0
,RELATED=*value*	*value:* Any valid macro keyword specification.

The explanation of the parameters is as follows:

ecb addr
specifies the address of the fullword event control block representing the event.

,comp code
specifies the completion code to be placed in the event control block upon completion.

,RELATED=*value*
specifies information used to self-document macro instructions by 'relating' functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
WAIT1    WAIT     1,ECB=ECB,RELATED=(RESUME1,
                  'WAIT FOR EVENT')
   .
   .
   .
RESUME1 POST      ECB,0,RELATED=(WAIT1,
                  'RESUME WAITER')
```

Note: Each of these macro instructions will fit on one line when coded, so there is no need for a continuation indicator.

Example 1

Operation: Signal event completion with a default completion code. POSTECB is the address of an ECB.

```
POST      POSTECB
```

Example 2

Operation: Signal event completion with a completion code of X'7FF'. POSTECB is the address of an ECB.

```
POST      POSTECB,X'7FF'
```

RETURN — Return Control

The RETURN macro instruction restores the control to the calling program and signals normal termination of the called program. The return of control is always made by executing a branch instruction using the address in register 14. Because the RETURN macro uses a BR 14 to pass control, it can be used only when the return is to a program that executes in the same addressing mode. The RETURN macro instruction can restore a designated range of registers, provide a return code in register 15, and flag the save area used by the called program.

If registers are to be restored, or if an indicator is to be placed into the save area, register 13 must contain the address of the save area, which must have the standard format.

The RETURN macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede RETURN.
RETURN	
b	One or more blanks must follow RETURN.

(*reg1*) (*reg1*,*reg2*)	*reg1* and *reg2:* decimal digits, and in the order 14, 15, 0 through 12.
,T	
,RC=*ret code*	*ret code:* decimal digit, symbol, or register (15). The maximum value is 4095.

The parameters are explained as follows:

(reg1)
(reg1,reg2)
specifies the register or range of registers to be restored from the save area pointed to by the address in register 13. If you omit this parameter, the contents of the registers are not altered. Do not code this parameter when returning control from a program interruption exit routine.

,T
causes the control program to flag the save area used by the called program. The low-order bit of word 4 of the save area is set to 1 after the registers have been loaded; this designates that a called program has executed a return to its caller. Do not specify this parameter when returning control from an exit routine.

,RC=*ret code*
specifies the return code to be passed to the calling program. If a symbol or decimal digit is coded, the return code is placed right-adjusted in register 15 before return is made; if register 15 is coded, the return code has been previously loaded into register 15 and the contents of register 15 are not altered or restored from the save area. (If you omit this parameter, the contents of register 15 are determined by the *reg1* and *reg2* parameters.)

Note: If register 15 is coded and a return code greater than 4095 (decimal) is passed, the results could be either an invalid return code in the message or invalid RC testing.

Example 1

Operation: Restore registers 14-12, flag the save area, and return with a code of 0.

```
RETURN  (14,12),T,RC=0
```

SAVE — Save Register Contents

The SAVE macro instruction stores the contents of the specified registers in the save area at the address contained in register 13. If you wish, you may specify an entry point identifier. Write the SAVE macro instruction only at the entry point of a program because the code resulting from the macro expansion requires that register 15 contain the address of the SAVE macro prior to its execution. Do not use the SAVE macro instruction in a program interruption exit routine.

The SAVE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede SAVE.
SAVE	
ƀ	One or more blanks must follow SAVE.
(reg1) *(reg1,reg2)*	*reg1* and *reg2:* decimal digits, and in the order 14, 15, 0 through 12.
, ,T	
,id name	*id name:* character string of up to 70 characters or as an *.

The parameters are explained as follows:

(reg1)
(reg1,reg2)
specifies the register or range of registers to be stored in the save area at the address contained in register 13. The registers are stored in words 4 through 18 of the save area.

,
,T
specifies that registers 14 and 15 are to be stored in word 4 and 5, respectively, of the save area. This parameter permits you to save two noncontiguous sets of registers.

If you specify both T and *reg2*, and *reg1* is any of registers 14, 15, 0, 1, or 2, all of registers 14 through the *reg2* value are saved.

,id name
specifies an identifier to be associated with the SAVE macro instruction. If an asterisk (*) is coded, the identifier is the *name* associated with the SAVE macro instruction, or, if the *name* field is blank, the control section name is used. The identifier aids in locating a program's save area in a dump. If the CSECT instruction name field is blank, the parameter is ignored.

Whenever a symbol or an asterisk is coded, the following macro expansion occurs:

- A count byte containing the number of characters in the identifier name is assembled four bytes following the address contained in register 15.
- The character string containing the identifier name is assembled starting at five bytes following the address contained in register 15.
- An instruction to branch around the count and identifier fields is assembled.

Example 1

Operation: Save registers 14-12, and associate the identifier with the CSECT name.

```
SAVE    (14,12),,*
```

SEGLD — Load Overlay Segment and Continue Processing

The SEGLD macro instruction loads the specified segment and any segments in its path that are not part of a path already in virtual storage. Control is not passed to the specified segment, but is returned to the instruction following the SEGLD macro instruction. Processing is overlapped with the loading of the segment. Refer to the *Linkage Editor and Loader* for details on overlay.

Note: This macro can be used only by callers in 24-bit addressing mode.

The SEGLD macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SEGLD.
SEGLD	
b	One or more blanks must follow SEGLD.
ext seg name	*ext seg name:* symbol.

The parameters are explained as follows:

ext seg name
specifies the name of a control section or an entry name in the required section. An exclusive reference is not allowed. The name does not have to be identified by an EXTRN statement.

Example 1

Operation: Cause the control program to load segment PGM54.

```
SEGLD   PGM54
```

SEGWT — Load Overlay Segment and Wait

The SEGWT macro instruction causes the control program to load the specified segment and any segments in its path that are not part of a path already in virtual storage. Control is not passed to the specified segment; control is not returned to the segment issuing the SEGWT macro instruction until the requested segment is loaded. Refer to the publication *Linkage Editor and Loader* for details on overlay operations. The SEGWT macro instruction cannot be used in an asynchronous exit routine.

Note: This macro can be used only by callers in 24-bit addressing mode.

The SEGWT macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SEGWT.
SEGWT	
b	One or more blanks must follow SEGWT.
ext seg name	*ext seg name:* symbol.

The parameters are explained as follows:

ext seg name
specifies the name of a control section or an entry name in the required section. An exclusive reference is not allowed. The name does not have to be identified by an EXTRN statement.

Example 1

Operation: Cause the control program to load segment PGMWT.

```
SEGWT   PGMWT
```

SETRP — Set Return Parameters

The SETRP macro instruction indicates the various requests that a recovery exit might make. Use it only if a system diagnostic work area (SDWA) was passed to the recovery exit. The macro instruction is valid only for ESTAE/ESTAI exits. (The SDWA mapping macro - IHASDWA - must be included in the routine that issues SETRP.)

The SETRP macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SETRP.
SETRP	
b	One or more blanks must follow SETRP.
,WKAREA=(*reg*)	*reg:* decimal digits 1-12. **Default:** WKAREA=(1)
,REGS=(*reg1*) ,REGS=(*reg1*,*reg2*)	*reg1:* decimal digits 0-12, 14 15. *reg2:* decimal digits 0-12, 14, 15. **Note:** If *reg1* and *reg2* are both specified, order is 14, 15, 0-12.
,DUMP=IGNORE ,DUMP=YES ,DUMP=NO	**Default:** DUMP=IGNORE
,DUMPOPT=*parm list addr*	*parm list addr:* RX-type address, or register (2) - (12). **Note:** This parameter may be specified only if DUMP=YES is specified above.
,REASON=*code*	*code:* any four-byte number specified in decimal (31-bit) or hexadecimal (32-bit).
,RC=0 ,RC=4 ,RC=16	**Default:** RC=0
,RETADDR=*retry addr*	*retry addr:* RX-type address, or register (2) - (12). **Note:** This parameter may be specified only if RC=4 is specified above. *reg info addr:* RX-type address, or register (2) - (12).
,RETREGS=NO ,RETREGS=YES ,RETREGS=YES,RUB=*reg info addr*	*reg info addr:* RX-type address, or register (2) - (12). **Default:** RETREGS=NO **Note:** This parameter may be specified only if RC=4 is specified above.
,FRESDWA=NO ,FRESDWA=YES	**Default:** FRESDWA=NO **Note:** This parameter may be specified only if RC=4 is specified above.
,COMPCOD=*comp code* ,COMPCOD=(*comp code*,USER) ,COMPCOD=(*comp code*,SYSTEM)	*comp code:* symbol, decimal digit, or register (2) - (12). **Default:** COMPCOD=(*comp code*,USER)

The parameters are explained as follows:

,WKAREA=(*reg*)
specifies the address of the SDWA passed to the recovery exit. If this parameter is omitted the address of the SDWA must be in register 1.

,REGS=(*reg1*)
,REGS=(*reg1*,*reg2*)
specifies the register or range of registers to be restored from the save area pointed to by the address in register 13. If REGS is specified, a branch on register 14 instruction will also be generated to return control to the control program. If REGS is not specified, the user must code his own return.

,DUMP=IGNORE
,DUMP=YES
,DUMP=NO
specifies that the dump option fields will not be changed (IGNORE), will be zeroed (NO), or will be merged with dump options specified in previous dump requests, if any (YES). If IGNORE is specified, a previous exit had requested a dump or a dump had been requested via the ABEND macro instruction, and the previous request will remain intact. If NO is specified, no dump will be taken.

,DUMPOPT=*parm list addr*
specifies the address of a parameter list that is valid for the SNAP macro instruction. The parameter list can be created by using the list form of the SNAP macro instruction, or a compatible list can be created. If the specified dump options include subpools for storage areas to be dumped, up to seven subpools can be dumped. Subpool areas are accumulated and wrapped, so that the eighth subpool area specified replaces the first. The TCB, DCB, and STRHDR options available on SNAP will be ignored if they appear in the parameter list. The TCB used will be the one for the task that suffered error. The DCB used will be one created by the control program and either SYSABEND, SYSMDUMP, or SYSUDUMP will be used as a DDNAME.

,REASON=*code*
specifies the reason code that the user wishes to pass to subsequent recovery exits.

,RC=0
,RC=4
,RC=16
specifies the return code the user exit routine sends to recovery processing to indicate what further action is required:

0 - Continue with termination, causes entry into previously specified recovery routine, if any.

4 - Retry using the retry address specified.

16 - Suppress further ESTAI/STAI processing (for ESTAI only).

,RETADDR=*retry addr*
specifies the address of the retry routine to which control is to be given.

,RETREGS=NO
,RETREGS=YES
,RETREGS=YES,RUB =*reg info addr*
specifies the contents of the registers on entry to the retry routine. If NO is specified or defaulted, only parameter registers (14-2) are passed; all others are unpredictable. If YES is specified, the contents of the SDWASRSV field will be used to initialize registers 0-14 when an FRR requests retry and registers 0-15 when an ESTAE requests a retry. For ESTAE exits, this field contains the registers at the last interruption of the RB level at which retry will occur. For ESTAI exits, the contents of SDWASRSV must be set by the user either before SETRP is issued or by use of the RUB parameter; any field not set will cause the corresponding register to contain 0 on entry to the retry routine.

RUB specifies the address of an area (register update block) that contains register update information. The data specified in this area will be moved into the SDWA and will be loaded into the general purpose registers on entry to the retry routine. The maximum length of the RUB is 66 bytes.

- The first two bytes represent the registers to be updated, register 0 corresponding to bit 0, register 1 corresponding to bit 1, and so on. The user indicates which of the registers are to be stored in the SDWA by setting the corresponding bits in these two bytes.

- The remaining 64 bytes contain the update information for the registers, in the order 0-15. If all 16 registers are being updated, this field consists of 64 bytes. If only one register is being updated, this field consists of only 4 bytes for that one register.

  For example, if only registers 4, 6, and 9 are being updated:

- Bits 4, 6, and 9 of the first two bytes are set.

- The remaining field consists of 12 bytes for registers 4, 6, and 9; the first 4 bytes are for register 4, followed by 4 bytes for register 6, and 4 final bytes for register 9.

,FRESDWA=NO
,FRESDWA=YES
specifies that the entire SDWA be freed (YES) or not be freed (NO) prior to entry into the retry routine.

,COMPCOD=*comp code*
,COMPCOD=(*comp code***,USER)**
,COMPCOD=(*comp code***,SYSTEM)**
specifies the user or system completion code that the user wishes to pass to subsequent recovery exits.

Example 1

Operation: Request continue with termination, suppress dumping, restore register 14 from the save area and pass control to the location it contains, contain the SDWA in the location addressed by register 3, and change the completion code to 10.

```
SETRP   RC=0,DUMP=NO,REGS=(14),WKAREA=(3),                X
        COMPCOD=(X'00A',USER)
```

Example 2

Operation: Retry using address X, take a dump before retry, use the contents of SDWASRSV to initialize the registers, free the SDWA before control is passed to the retry address, and restore registers 14-12.

```
SETRP   RC=4,RETREGS=YES,DUMP=YES,FRESDWA=YES,            X
        REGS=(14,12),RETADDR=X
```

SNAP — Dump Virtual Storage and Continue

The SNAP macro instruction dumps some or all of the storage assigned to the current job step. You can also dump some or all of the control program fields. The SNAP macro instruction causes the specified storage to be displayed in the addressing mode of the caller.

You must provide a data control block and issue an OPEN macro instruction for the data set before any SNAP macro instructions are issued. The DCB macro instruction *must* contain the following parameters:

```
DSORG=PS,RECFM=VBA,MACRF=(W),BLKSIZE=nnn,LRECL=xxx,
and DDNAME=any name but SYSABEND, SYSMDUMP or SYSUDUMP
```

The DCB and TCB must reside in 24-bit addressable storage. All other parameters can reside above 16 Mb if the issuer is executing in 31-bit addressing mode.

If a standard dump of 120 characters per line is requested, BLKSIZE must be either 882 or 1632, and LRECL must be 125. (The data control block and the DCB macro instruction are described in *Data Management Services Guide* and *Data Management Macro Instructions.*) A high-density dump printed on a 3800 Printing Subsystem has 204 characters per line. To obtain a high-density dump, you must code CHARS=DUMP on the DD statement describing the dump data set. The BLKSIZE must be either 1470 or 2724, and the LRECL must be 209. You can also code CHARS=DUMP on the DD statement describing a dump data set that will not be printed immediately. If you specify CHARS=DUMP and the output device is not a 3800, print lines are truncated and print data is lost. If you open a SNAP data set in a problem program that will be processed by the system loader, your problem program must close the data set.

There are three ways to obtain a dump:

1. Spool the dump by specifying SYSOUT=x on the DD statement. The dump is printed without a separate job but is deferred until after the job ends.

2. Select a tape or direct access device. This method requires a separate job step to print the dump. This method might be used if the dump is to be printed more than once.

3. Select a printer on the DD statement. This method is almost never used because the printer cannot be used by anyone else for the duration of the job step.

Both NUC and ALLVNUC are valid. Only ALLVNUC gives you the whole virtual nucleus.

The standard form of the SNAP macro instruction is written as follows:

name	*name:* symbol. **Begin** *name* in column 1.
b	One or **more blanks** must precede SNAP.
SNAP	
b	One or **more blanks** must follow SNAP.
DCB=*dcb addr*	*dcb addr:* **A-type address**, or register (2) - (12).
,**TCB=***tcb addr*	*tcb addr:* **A-type address**, or register (2) - (12).
,**ID=***id nmbr*	*id nmbr:* symbol, decimal digit, or register (2) - (12). **Value range:** 0-255
,**SDATA=ALL** ,**SDATA=(***sys data code***)**	*sys data code:* any combination of the following, separated by commas. If you specify only one code, you do not need the parentheses. NUC CB ERR SQA Q IO LSQA TRT ALLVNUC PCDATA SWA DM SUM
,**PDATA=ALL** ,**PDATA=(***prob data code***)**	*prob data code:* any combination of the following, separated by commas. If you specify only one code, you do not need the parentheses. PSW REGS SA or SAH JPA or LPA or ALLPA SPLS SUBTASKS
,**STORAGE=(***strt addr,end addr***)** ,**LIST=***list addr*	*strt addr:* A-type address, or register (2) - (12). *end addr:* A-type address, or register (2) - (12). *list addr:* A-type address, or register (2) - (12). **Note:** One or more pairs of addresses may be specified, separated by commas. For example: STORAGE=(*strt addr,end addr,strt addr,end addr*)
,**STRHDR=***(hdr addr)* ,**STRHDR=***hdr list addr*	*hdr addr:* A-type address, or register (2) - (12). **Note:** *hdr addr* is one or more addresses separated by commas. If you specify only one header address as an A-type address, you do not need the parentheses. If you specify one or more registers, then you must code double parentheses (one set enclosing each register and one set enclosing the list of registers). If STRHDR=(*hdr addr*) is specified, then STORAGE must also be specified. *hdr list addr:* A-type address, or register (2) - (12). **Note:** If STRHDR=*hdr list addr* is specified, then LIST must also be specified.
,**SUBPLST=***sbp list addr*	*sbp list addr:* A-type address, or register (2) - (12).

The parameters are explained as follows:

DCB=*dcb addr*
specifies the address of a previously opened data control block for the data set that is to contain the dump.

Note: DCB must reside in 24-bit addressable storage.

,TCB=*tcb addr*

specifies the address of a fullword on a fullword boundary containing the address of the task control block for a task of the current job step. If omitted, or if the fullword contains 0, the dump is for the active task. If a register is designated, the register can contain 0 to indicate the active task, or can contain the address of a TCB.

Note: TCB must reside in 24-bit addressable storage.

,ID=*id nmbr*

specifies the number that is to be printed in the identification heading with the dump. If the number specified is not in the acceptable value range, it will not be printed properly in the heading.

,SDATA=ALL
,SDATA=(*sys data code***)**

specifies the system control program information to be dumped:

ALL	All of the SDATA options except ALLVNUC (The read-only portion of the nucleus is not included in the dump unless ALLVNUC is also specified as an option.)
NUC	The PSA, SQA, LSQA, and the read/write portion of the nucleus (if the entire nucleus is required, specify the ALLVNUC option.) *Note:* The CVT will be included if this option is specified.
SQA	The system queue area (subpools 226, 239, and 245). *Note:* Subpools 229 and 230 will only be dumped for the current task.
LSQA	The local system queue area and subpools 229 and 230.
SWA	The scheduler work area related to the task (subpools 236 and 237).
CB	The control blocks for the task.
Q	The global resource serialization control blocks for the task.
TRT	The GTF trace and system trace data. If system tracing is active and the requestor is authorized, all system trace entries for all address spaces are included in the dump. Unauthorized requestors obtain those system trace entries, after the job-start time stamp in the ASCB, for their current address space. If GTF tracing is active, only the GTF trace entries for the current address space are included in the dump.
DM	Data management control blocks for the task.
ERR	Recovery/termination control blocks for the task. These control blocks summarize information that describes abnormal terminations of the task.
IO	Input/Output supervisor control blocks for the task.
ALLVNUC	The entire virtual nucleus, the PSA, LSQA, and SQA. (The NUC option will not dump the read-only section of the nucleus.) If the SNAP parameter list is used for a SYSMDUMP, the ALLVNUC option is converted to ALLNUC on the SVC dump parameter list. *Note:* The CVT is included if this option is specified.
PCDATA	Program call information for the task.

The SUM option is valid for an abending task or on a list form of the SNAP macro instruction pointed to by the DUMPOPT keyword of the ABEND or SETRP macro instructions. The option SUM causes the dump to contain a summary dump. If SUM is the only option requested, the dump contains a dump header, control blocks, and the

other areas listed below. The header information, which is provided for all ABEND dumps, consists of the following information:

- The dump title
- The ABEND code and program status word (PSW) at the time of the error
- If the PSW contains the address of an active load module:
  - The name and PSW address of the load module in error
  - The offset, into the load module, at which the error occurred

The following control blocks and areas are also included in the dump:

- The control blocks dumped for the CB option
- The error control blocks (RTM2WAs and SCBs)
- The save areas
- The registers at the time of the error
- The contents of the load module (if the PSW contains the address of an active load module)
- The module pointed to by the last PRB (if it can be found)
- 1K of storage before and after the addresses pointed to by the PSW and the registers at the time of the error.

  Note: This storage will only be dumped if the caller is authorized to obtain it. The storage is printed by ascending storage addresses with duplicate addresses removed.

- System trace entries after the job-start time stamp in the ASCB for the current address space.

  Note: The GTF trace records are not included.

If other options are specified with SUM, the summary dump is dispersed throughout the dump. See the topic "SNAP Dumps" for an example of how this is done.

,PDATA=ALL
,PDATA=(*prob data code*)
specifies the problem program information to be dumped:

ALL	All of the following fields.
PSW	Program status word when the SNAP or ABEND macro instruction was issued.
REGS	Contents of the floating-point registers and general-purpose registers when the SNAP or ABEND macro instruction was issued. Also, contents of the vector registers, vector status register, and the vector mask register when the SNAP or ABEND macro instruction was issued for any task that uses the Vector Facility.
SA	Save area linkage information, program call linkage information, and a back trace through save areas.

SAH — Save area linkage information and program call linkage information.

JPA — Contents of job pack area.

LPA — Contents of active link pack area for the requested task.

ALLPA — Contents of job pack area and active link pack area for the requested task.

SPLS — All virtual storage subpools (0-127,252).

SUBTASKS — The designated task and the program data information for all of its subtasks.

,STORAGE=(*strt addr,end addr***)**
,LIST=*list addr*

specifies one or more pairs of starting and ending addresses or a list of starting and ending addresses of areas to be dumped. Each starting address is rounded down to a fullword boundary; each ending address is rounded up to a fullword boundary. The area is then dumped in fullword increments. Callers executing in either 24-bit or 31-bit addressing mode must set the high-order bit of the fullword containing the last address in this list to 1. Callers executing in 31-bit addressing mode must ensure that this bit is cleared in all other addresses in the list because SNAP processing truncates the list at the first address that contains a 1 in the high order bit.

,STRHDR=(*hdr addr***)**
,STRHDR=*hdr list addr*

specifies one or more header addresses or the address of a list of header addresses. Each header address must be the address of a one byte header length field, which is followed by the text of the header. The header has a maximum length of 100 characters.

If the STORAGE parameter was specified, the STRHDR (storage header) value must be one or more header addresses. The number of pairs of starting and ending addresses specified for STORAGE must be the same as the number of header addresses specified for STRHDR. If a header is not desired for a storage area, a comma must be used to indicate its absence.

If the LIST parameter was specified, the STRHDR value must be the address of a list of header addresses. The list of addresses must begin on a fullword boundary, and the high order bit of the fullword containing the last address of the list must be set to 1. The number of pairs of starting and ending addresses supplied with the LIST parameter must be the same as the number of addresses in the list supplied with STRHDR. If a header is not desired for a storage area, the STRHDR list must contain a zero address to indicate its absence.

,SUBPLST=*sbp list addr*

specifies the address of a list of subpool numbers to be dumped. Each entry in the list must be a two-byte entry and must specify a valid subpool number. The first halfword of the list must contain the number of subpools in the list and must be on a fullword boundary. If you specify an invalid subpool number or a subpool number for which you do not have authorization, the number is skipped and you receive a comment in the dump output indicating the error. If a subpool contains 4k blocks of data that are mapped from a linear data set, the dump includes only the blocks that have changed since the last DIV SAVE function was invoked.

Note: A maximum of seven subpool numbers is permitted on the list form of the SNAP macro instruction pointed to by the DUMPOPT keyword of ABEND or SETRP.

Control is returned to the instruction following the SNAP macro instruction. When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion.
04	Data control block was not open, or an invalid page exception occurred during the validity check of the DCB parameters.
08	Task control block address was not valid, an invalid page reference occurred during the validity check of the TCB address, a subtask is a job step task, sufficient storage was not available, or the READ for JFCB or JFCBE failed. In all cases, the dump is canceled. (Message IEA997I is issued when the READ for JFCB or JFCBE fails.)
0C	Data control block type (DSORG, RECFM, MACRF, BLKSIZE, or LRECL) was incorrect, or the DCB's BLKSIZE and/or LRECL were not compatible with the dump format options specified on the dump-related DD statement.

Example 1

Operation: Dump the storage ranges pointed to by register 9, and dump all PDATA and SDATA options.

```
SNAP    DCB=(8),TCB=(5),PDATA=ALL,SDATA=ALL,LIST=(9)
```

Example 2

Operation: Dump the storage ranges pointed to by register 9, and dump only the trace table and enqueue control blocks.

```
SNAP    DCB=(8),TCB=(5),ID=4,LIST=(9),SDATA=(TRT,Q)
```

Example 3

Operation: Dump storage area 1000-2000 with no header, and dump storage area 3000-4000 with a header of 'USER LABEL ONE'. The comma specified in the value for STRHDR indicates that no header is wanted for storage area 1000-2000.

```
        SNAP    DCB=(8),STORAGE=(1000,2000,3000,4000),    X
                STRHDR=(,L1)
        .
        .
        .
L1      DC      AL1(L'HDR1)
HDR1    DC      C'USER LABEL ONE'
```

Example 4

Operation: Dump storage area 1000-1999 with a header of 'LABEL ONE' and dump storage area 3000-3999 with a header of 'LABEL TWO'.

```
        SNAP    DCB=(8),LIST=X,STRHDR=L1
        .
        .
        .
X       DC  A(1000)           Start address
        DC  A(1999)           End address
        DC  A(3000)           Start address
        DC  X'80'             End of list indicator
        DC  AL3(3999)         End address
L1      DC  A(HDR1)           Address of length label for
                              header one
        DC  X'80'             End of list
        DC  AL3(HDR2)         Address of length label for
                              header two
HDR1    DC  AL1(L'HDR1A)      Length of header one
HDR1A   DC  C'LABEL ONE'      Header one
HDR2    DC  AL1(L'HDR2A)      Length of header two
HDR2A   DC  C'LABEL TWO'      Header two
```

Example 5

Operation: Dump subpool 0, 1, and 2 storage related to the current TCB.

```
SNAP DCB=XYZ,TCB=0,SUBPLST=SUBADDR
            .
            .
            .
SUBADDR DS OF           Fullword boundary
        DC X'0003'      Number of entries in the list
        DC X'0000'      Subpool 0
        DC X'0001'      Subpool 1
        DC X'0002'      Subpool 2
```

SNAP (List Form)

Use the list form of the SNAP macro to construct a control program parameter list. You can specify any number of storage addresses using the STORAGE parameter. Therefore, the number of starting and ending address pairs in the list form of SNAP must be equal to the maximum number of addresses specified in any execute form of the macro, or a DS instruction must immediately follow the list form to allow for the maximum number of addresses.

The list form of the SNAP macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SNAP.
SNAP	
b	One or more blanks must follow SNAP.

DCB=*dcb addr*	*dcb addr:* A-type address.
,ID=*id nmbr*	*id nmbr:* symbol or decimal digit. **Value range:** 0-255
,SDATA=ALL ,SDATA=(*sys data code*)	*sys data code:* any combination of the following, separated by commas. If you specify only one code, you do not need parentheses. NUC CB ERR SQA Q IO LSQA TRT ALLVNUC PCDATA SWA DM SUM
,PDATA=ALL ,PDATA=(*prob data code*)	*prob data code:* any combination of the following, separated by commas. If you specify only one code, you do not need parentheses. PSW REGS SA or SAH JPA or LPA or ALLPA SPLS SUBTASKS
,STORAGE=(*strt addr,end addr*) ,LIST=*list addr*	*strt addr:* A-type address. *end addr:* A-type address. *list addr:* A-type address. **Note:** One or more pairs of addresses may be specified, separated by commas. For example: STORAGE=(*strt addr,end addr,strt addr,end addr*)
,STRHDR=(*hdr addr*)	*hdr addr:* A-type address. **Note:** hdr addr is one or more addresses separated by commas. If you specify only one header address, you do not need the parentheses. If STRHDR=(*hdr addr*) is specified, then STORAGE must also be specified.
,STRHDR=*hdr list addr*	*hdr list addr:* A-type address. **Note:** If STRHDR=*hdr list addr* is specified, then LIST must also be specified.
,SUBPLST=*sbp list addr*	*sbp list addr:* A-type address.
,MF=L	

The parameters are explained under the standard form of the SNAP macro instruction, with the following exception:

,MF = L
specifies the list form of the SNAP macro instruction.

SNAP (Execute Form)

A remote control program parameter list is referred to and can be modified by the execute form of the SNAP macro instruction.

If you code only the DCB, ID, MF, or TCB parameters in the execute form of the macro instruction, the bit settings in the parameter list corresponding to the SDATA, PDATA, LIST, and STORAGE parameters are not changed. However, if you code the SDATA, PDATA, or LIST parameters, the bit settings for the coded parameter from the previous request are reset to zero, and only the areas requested in the current macro instruction are dumped.

The execute form of the SNAP macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SNAP.
SNAP	
b	One or more blanks must follow SNAP.
DCB=*dcb addr*	*dcb addr:* RX-type address, or register (2) - (12).
,TCB=*tcb addr* ,TCB='S'	*tcb addr:* RX-type address, or register (2) - (12).
,ID=*id nmbr*	*id nmbr:* symbol, decimal digit or register (2) - (12). **Value range:** 0-255
,SDATA=ALL ,SDATA=(*sys data code*)	*sys data code:* any combination of the following, separated by commas. If you specify only one code, you do not need parentheses. NUC CB ERR SQA Q IO LSQA TRT ALLVNUC PCDATA SWA DM SUM
,PDATA=ALL ,PDATA=(*prob data code*)	*prob data code:* any combination of the following, separated by commas. If you specify only one code, you do not need parentheses. PSW REGS SA or SAH JPA or LPA or ALLPA SPLS SUBTASKS
,STORAGE=(*strt addr,end addr*)	*strt addr:* RX-type address, or register (2) - (12).
,LIST=*list addr*	*end addr:* RX-type address, or register (2) - (12). *list addr:* RX-type address, or register (2) - (12). **Note:** One or more pairs of addresses may be specified, separated by commas. For example: STORAGE=(*strt addr,end addr,strt addr,end addr*)
,STRHDR=(*hdr addr*)	*hdr addr:* RX-type address, or register (2) - (12). **Note:** hdr addr is one or more addresses separated by commas. If you specify only one header address as an RX-type address, you do not need the parentheses. If you specify one or more registers, then you must code double parentheses (one set enclosing each register and one set enclosing the list of registers). If STRHDR=*(hdr addr)* is specified, then STORAGE must also be specified.
,STRHDR=*hdr list addr*	*hdr list addr:* RX-type address, or register (2) - (12). **Note:** If STRHDR=*hdr list addr* is specified, then LIST must also be specified.
,SUBPLST=*sbp list addr*	*sbp list addr:* RX-type address, or register (2) - (12).
,MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) or (2) - (12).

The parameters are explained under the standard form of the SNAP macro instruction, with the following exceptions:

,TCB='S'
specifies the task control block of the active task.

Note: TCB='S' causes a dump of the active task if this is the first use of the list form of the SNAP macro instruction or if the TCB specified on a previous execute form of the SNAP macro instruction was the current TCB or TCB='S'.

,MF=(E,*ctrl addr*)
specifies the execute form of the SNAP macro instruction using a remote control program parameter list.

SPIE — Specify Program Interruption Exit

The SPIE macro instruction specifies the address of an interruption exit routine and the program interruption types that are to cause the exit routine to be given control. If the program interruption types specified can be masked, the corresponding program mask bit in the PSW (program status word) is set to 1. If a maskable interruption is not specified, the corresponding bit in the PSW is set to 0.

Only callers in 24-bit addressing mode can issue the SPIE macro instruction. If a caller in 31-bit addressing mode issues a SPIE macro instruction, the caller is abended with a system completion code of X'30E'. Callers in 31-bit addressing mode must use the ESPIE macro instruction, which performs the same function as the SPIE macro instruction for callers in both 24-bit and 31-bit addressing mode. The ESPIE macro instruction is described in Part II of this book. For additional information concerning the relationship between the SPIE and the ESPIE macro instructions, see the section on program interruption processing, in Part I.

Each succeeding SPIE macro instruction completely overrides any previous SPIE macro instruction specifications for the task. The specified exit routine is given 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 issued the SPIE macro instruction terminates. If a caller attempts to delete a SPIE environment established under a previous RB, the caller is abended with a system completion code of X'46D'.

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 that issued 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

A PICA (program interruption control area) is created as part of the expansion of SPIE. The PICA contains the exit routine's address and a code indicating the interruption types specified in SPIE.

If a SPIE environment was active, the SPIE service routine returns the address of the previous PICA in register 1; if an ESPIE environment was active, the SPIE service routine returns the address of a fake PICA in register 1. The contents of the fake PICA are unpredictable. If no SPIE/ESPIE environment was active, the service routine returns a zero.

For more information on the SPIE macro and its control blocks, see the section on program interruption processing.

The standard form of the SPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SPIE.
SPIE	
b	One or more blanks must follow SPIE.
exit addr,(interrupts)	*exit addr:* A-type address, or register (2) - (12). *interrupts:* decimal digits, 1-15, expressed as: single values: (2,3,4,7,8,9,10) ranges of values: ((2,4),(7,10)) combinations: (2,3,4,(7,10))

The parameters are explained as follows:

exit addr,(interrupts)
specifies the address of the exit routine to be given control when a program interruption of the type specified occurs. The interruption types are:

Number	Interruption Type
1	Operation
2	Privileged operation
3	Execute
4	Protection
5	Addressing
6	Specification
7	Data
8	Fixed-point overflow (maskable)
9	Fixed-point divide
10	Decimal overflow (maskable)
11	Decimal divide
12	Exponent overflow
13	Exponent underflow (maskable)
14	Significance (maskable)
15	Floating-point divide

Notes:

1. *If an exit address is zero or no parameters are specified, the SPIE environment is cancelled.*

2. *If a program interruption type is maskable, the corresponding bit is set to 1 when specified and to 0 when not specified. Interruption types that are not maskable and not specified above are handled by the control program, which forces an abend with the program check as the completion code. If an ESTAE-type recovery routine is also active, the SDWA indicates a system-forced abnormal termination. The registers at the time of the error are those of the control program.*

3. *As shown in the table above, interruption types can be designated as one or more single numbers, as one or more pairs of numbers (designating ranges of values), or as any combination of the two forms. For example, (4,8) indicates interruption types 4 and 8; ((4,8)) indicates interruption types 4 through 8.*

4. *For both ESPIE and SPIE — If you are using vector instructions and an exception of 8, 12, 13, 14, or 15 occurs, your recovery routine can check the exception extension code (the first byte of the two-byte interruption code in the EPIE or PIE) to determine whether the exception was a vector or scalar type of exception.*

Example 1

Operation: Give control to an exit routine for interruptions 1, 5, 7, 8, 9, and 10. DOITSPIE is the address of the SPIE exit routine.

```
SPIE   DOITSPIE,(1,5,7,(8,10))
```

SPIE (List Form)

Use the list form of the SPIE macro instruction to construct a control program parameter list in the form of a program interruption control area.

The list form of the SPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SPIE.
SPIE	
b	One or more blanks must follow SPIE.
exit addr, (*interrupts*), MF=L	*exit addr:* A-type address. *interrupts:* decimal digits, 1-15, expressed as: single values: (2,3,4,7,8,9,10) ranges of values: ((2,4),(7,10)) combinations: (2,3,4,(7,10))

The parameters are explained under the standard form of the SPIE macro instruction, with the following exception:

MF=L
specifies the list form of the SPIE macro instruction.

SPIE (Execute Form)

A remote control program parameter list, the program interruptions control area (PICA) is used in, and can be modified by, the execute form of the SPIE macro instruction. The PICA can be generated by the list form of SPIE, or you can use the address of the PICA returned in register 1 following a previous SPIE macro instruction. If this macro instruction is being issued to reestablish a previous SPIE environment, code only the MF parameter.

The address of the remote control program parameter list associated with any previous SPIE environment is returned by the SPIE macro instruction.

The execute form of the SPIE macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SPIE.
SPIE	
b	One or more blanks must follow SPIE.
exit addr, (*interrupts*),	*exit addr:* RX-type address, or register (2) - (12). *interrupts:* decimal digits, 1-15, expressed as: single values: (2,3,4,7,8,9,10) ranges of values: ((2,4),(7,10)) combinations: (2,3,4,(7,10))
MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) or (2) - (12).

The parameters are explained under the standard form of the SPIE macro instruction, with the following exception:

MF=(E,*ctrl addr*)
specifies the execute form of the SPIE macro instruction using a remote control program parameter list.

Note: If SPIE is coded with a 0 as the control address, the SPIE environment is cancelled.

SPLEVEL — SET and TEST Macro Level

Specific macro instructions supplied in the MVS/XA macro library are identified as downward incompatible (to MVS/370 System Product Version 1 Release 3). Unless the user takes specific action, these macros generate downward incompatible statements. It is possible to cause the generation of downward compatible expansions of these macros by using the SPLEVEL macro instruction. The downward incompatible macro instructions interrogate a global symbol (set by SPLEVEL) during assembly to determine the type of expansion to be generated. See the topic "Selecting the Macro Level" for additional information concerning the downward incompatible macro instructions and *Assembler H Version 2 Application Programming: Language Reference* for information about global set symbols.

The SPLEVEL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SPLEVEL.
SPLEVEL	
b	One or more blanks must follow SPLEVEL.

SET=*n* SET TEST	*n:* 1 or 2. **Default:** SET=2

The parameters are explained as follows:

SET=*n*
SET
TEST
specifies whether the macro level is being set or tested.

If SET=*n* is specified, SPLEVEL processing sets a global set symbol equal to *n*, where *n* must be 1 or 2. If a user codes one of the downward incompatible macros, one of the following macro expansions is generated:

- the MVS/370 (System Product Version 1 Release 3) macro expansion if *n*=1
- the MVS/XA macro expansion if *n*=2

If SET is specified without *n*, the SPLEVEL routine uses the default value, which is 2, unless the installation has changed the default.

The TEST option is used to determine the macro level that is in effect. The results of the test request are returned to the user in the global set symbol, &SYSSPLV, which is defined by "GBLC &SYSSPLV." If TEST is specified and if SPLEVEL SET has not been issued during this assembly, SPLEVEL processing puts the default value into the global set symbol. If SPLEVEL SET has been issued, the previous value of *n* or the default value is already in the global set symbol.

Example 1

Operation: Select the MVS/370 version of a specific downward incompatible macro instruction.

```
SPLEVEL SET=1
```

Example 2

Operation: Select the MVS/XA version of a specific downward incompatible macro instruction.

```
SPLEVEL SET=2
```

STATUS — Change Subtask Status

The STATUS macro instruction changes the dispatchability status of one or all of a program's subtasks. For example, use the STATUS macro instruction to restart subtasks that were stopped when an attention exit routine was entered.

The STATUS macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede STATUS.
STATUS	
b	One or more blanks must follow STATUS.
START STOP	
,TCB=*tcb addr*	*tcb addr:* RX-type address, or register (2) - (12).
,RELATED=*value*	*value:* Any valid macro keyword specification.

The parameters are explained as follows:

START
STOP
specifies that the START or STOP count in the task control block specified in the TCB parameter will be decreased (for START) or increased (for STOP) by 1. If the TCB parameter is not coded, the count is decreased/increased by 1 in the task control blocks for all the subtasks of the originating task.

Note: This parameter does not assure that the subtask(s) is stopped when control is returned to the issuer. A subtask can have a "stop deferred" condition that would cause that particular subtask to remain dispatchable until stops are no longer deferred. In an MP environment, it would be possible to have a task issue the STATUS macro with the STOP parameter and resume processing while the subtask (for which the STOP was issued) is re-dispatched to another processor.

,TCB=*tcb addr*
specifies the address of a fullword on a fullword boundary containing the address of the task control block that is to have its START/STOP count adjusted. (If a register is specified, however, the address is of the TCB itself.) If this parameter is not coded, the count is adjusted in the task control blocks for all the subtasks of the originating task.

Note: TCB must reside in 24-bit addressable storage.

,RELATED=*value*
specifies information used to self-document macro instructions by "relating" functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
STAT1    STATUS    STOP,TCB=YOURTCB,RELATED=(STAT2,
                   'STOP A SUBTASK')
  .
  .
  .
STAT2    STATUS    START,TCB=YOURTCB,RELATED=(STAT1,
                   'START A SUBTASK')
```

Note: Each of these macro instructions will fit on one line when coded, so there is no need for a continuation indicator.

Example 1

Operation: Stop all subtasks.

```
STATUS    STOP
```

Example 2

Operation: Stop a specific subtask. WHERETCB is a fullword specifying the address of a subtask TCB.

```
STATUS    STOP,TCB=WHERETCB
```

Example 3

Operation: Start a specific subtask. WHERETCB is a fullword specifying the address of a subtask TCB.

```
STATUS    START,TCB=WHERETCB
```

STIMER — Set Interval Timer

The STIMER macro instruction sets a timer to a specified time interval (less than or equal to 24 hours) or to an interval that will expire at a specified time of day (not to exceed 24:00:00:00). An optional asynchronous timer completion exit is given control when the time interval expires; if no asynchronous timer completion routine is specified, no indication that the time interval has expired is provided. Only one time interval per task is in effect at a time, using STIMER, however, using STIMERM in conjunction with STIMER allows 17 separate intervals to be associated with a task. A second STIMER macro instruction issued before the first time interval expires overrides the first interval and exit routine. If a timer exit routine issues an STIMER macro instruction specifying the same exit routine, an infinite loop might result.

The time interval may be a 'real-time interval' (measured continuously in real time via the clock comparator), or a 'task time interval' (measured, only while the task is in execution, via the CPU timer). If a real time interval is specified, the task may elect to either continue (REAL) or suspend (WAIT) execution during the interval. If the task elects to continue execution, it may optionally specify an exit routine to be given control on completion of the time interval. If the task elects to suspend execution, it is restarted at the next sequential instruction on completion of the time interval. If a task time interval is specified, the task must continue. It may optionally specify an exit routine to be given control on completion of the interval.

This macro can be assembled compatibly between MVS/XA and MVS/370 through the use of the SPLEVEL macro instruction. Default processing will result in an expansion of the macro that operates only with MVS/XA. See the topic "Selecting the Macro Level" for additional information.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction.

The STIMER macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede STIMER.
STIMER	
ƀ	One or more blanks must follow STIMER.
REAL REAL,*exit rtn addr* TASK TASK,*exit rtn addr* WAIT	*exit rtn addr:* RX-type address, or register (0) or (2) - (12).
,BINTVL=*stor addr* ,DINTVL=*stor addr* ,GMT=*stor addr* ,MICVL=*stor addr* ,TOD=*stor addr* ,TUINTVL=*stor addr*	*stor addr:* RX-type address, or register (1) or (2) - (12). **Note:** The GMT and TOD parameters must not be specified with TASK above.
,ERRET=*err rtn addr*	*err rtn addr:* RX-type address or register (2) - (12).

The parameters are explained as follows:

REAL
REAL,*exit rtn addr*
TASK
TASK,*exit rtn addr*
WAIT

specifies whether the timer interval is a real-time interval (REAL or WAIT) or a task-time interval (TASK). You must specify one or more of these parameters.

For REAL, the interval is decreased continuously. If the TOD or GMT parameter is coded, the interval expires at the indicated time of day.

For TASK, the interval is decreased only when the associated task is active.

For WAIT, the interval is decreased continuously. The task is to be placed in the wait condition until the interval expires.

The *exit rtn addr* is the address of the timer completion exit routine to be given control after the specified time interval expires. The routine does not get control immediately when the interval completes, but at some time after the interval completes, depending on the system's work load and the relative dispatching priority of the associated task. The routine must be in virtual storage when it is required. The exit routine receives control in the same addressing mode that the caller had when the caller issued the STIMER macro. The contents of the registers when the exit routine is given control are as follows:

Register	Contents
0 - 1	Control program information.
2 - 12	Unpredictable.
13	Address of a control-program-provided save area.
14	Return address (to the control program).
15	Address of the exit routine.

The exit routine is responsible for saving and restoring registers. The exit routine executes as a subroutine, and must return control to the control program. Although timing services allows only one active time interval for a task, it does not serialize the use of an asynchronous timer completion exit routine.

,BINTVL=*stor addr*
,DINTVL=*stor addr*
,GMT=*stor addr*
,MICVL=*stor addr*
,TOD=*stor addr*
,TUINTVL=*stor addr*

requests that the time be returned. You must specify one or more of these parameters.

For BINTVL, the address is in virtual storage containing the time interval. The time interval is presented as an unsigned 32-bit binary number; the low-order bit has a value of 0.01 second.

For DINTVL, the address is a doubleword on a doubleword boundary in virtual storage containing the time interval. The time interval is presented as zoned decimal digits of the form:

HHMMSSth, where:

HH	is hours (24-hour clock)
MM	is minutes
SS	is seconds
t	is tenths of seconds
h	is hundredths of seconds

For GMT, the address is an 8-byte area containing the Greenwich mean time at which the interval is to be completed. The time is presented as zoned decimal digits of the form HHMMSSth, as described above under DINTVL.

For MICVL, the address is a doubleword on a doubleword boundary containing the time interval. The time interval is represented as an unsigned 64-bit binary number; bit 51 is the low-order bit of the interval value and equivalent to 1 microsecond.

For TOD, the address is a doubleword on a doubleword boundary containing the time of day at which the interval is to be completed. The time of day is presented as zoned decimal digits of the form HHMMSSth, as described above under DINTVL.

For TUINTVL, the address is a fullword on a fullword boundary containing the time interval. The time interval is presented as an unsigned 32-bit binary number; the low-order bit has a value of one timer unit (approximately 26.04166 microseconds).

Note: For the DINTVL, GMT, and TOD parameters, the zoned decimal digits are not checked for validity. Thus, the specification of invalid digits can result in an ABEND 0C7, or a time interval different from that desired.

,ERRET = *err rtn addr*

specifies the address of the routine to be given control when the STIMER function cannot be performed because of damaged clocks. The STIMER macro will test the return code and give control to the specified routine for a non-zero value. The register contents when the routine is given control are:

Register	Contents
0 - 1	unpredictable
2 - 14	unchanged
15	return code

If the caller does not specify ERRET, then the STIMER function will return only on successful completion (return code 0). If ERRET is not specified, failure due to damaged clock(s) will result in the abnormal termination of the caller.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion.
08	Damaged clocks.

Notes:

1. *The time interval specified by an STIMER macro instruction has no relation to the time interval specified in an EXEC statement.*

2. *If the optional exit routine address and WAIT are not specified, no indication of completion of the time interval is provided.*

3. *The TTIMER and CPUTIMER macro instructions provide a facility for determining the remaining time interval associated with STIMER.*

4. *The STIMER macro instruction should not be issued while a BTAM OPEN or LINE OPEN operation is in progress, since the BTAM OPEN LINE routines also use STIMER. STIMER should not be issued before invoking dynamic allocation because dynamic allocation can also issue STIMER.*

5. *Specifying a time interval greater than 24 hours and DINTVL (without TOD or GMT), BINVTL, MICVL, or TUINVL causes the time interval to be set to 24 hours.*

6. *Specifying a time interval greater than 24 hours and DINTVL with TOD or GMT, causes a 12F abend.*

The priorities of other tasks in the system can also affect the accuracy of the time interval measurement. If you code REAL or WAIT, the interval is decreased continuously and can expire when the task is not active. After the time interval expires, assuming the task is not in the wait condition for any other reasons, the task is placed in the ready condition and competes for control with the other ready tasks in the system. The additional time required before the task becomes active depends on the relative dispatching priority of the task.

Example 1

Operation: Request the user's asynchronous exit routine, located at location EXIT, to receive control after the number of hundredths of seconds specified at INTVLONG has elapsed in real time.

```
STIMER   REAL,EXIT,BINTVL=INTVLONG
```

STIMERM — Set, Test, Cancel Multiple Interval Timer

The STIMERM macro

- Sets a timer to a specified time interval (SET parameter),
- Tests the remaining time interval for a timer request (TEST parameter), and
- Cancels a specific timer request (CANCEL parameter).

The SET request sets a timer to a specified time interval (less than 24 hours) or to an interval that will expire at a specified time of day (not to exceed 24 hours). Up to sixteen STIMERM requests per task may be in effect at a time. Note that this limit of sixteen does not include time intervals established through the STIMER macro or the DIE function.

The time interval is a real-time interval, measured continuously. The task can continue (WAIT=NO) or suspend execution (WAIT=YES). If the task continues execution, it can pass control to an exit routine (EXIT parameter) when the time interval is complete. If you specify an exit routine, the task can optionally pass a parameter to the exit routine (PARM parameter). The task grants control to the optional asynchronous timer completion exit when the time interval expires. If the task did not specify either an asynchronous timer completion routine or WAIT=YES, the task will receive no indication that the time interval has expired.

The TEST request tests the remaining time interval for a timer requested via the SET parameter. The ID parameter identifies the particular timer request to be tested and must be established by the current task.

The CANCEL request cancels a specific timer request or all of the current task's timer requests that were established via the SET parameter. The ID= parameter identifies the timer requests to be cancelled. If the macro cancels a specific timer request, it may return the remaining time interval for that request to a storage area designated by the TU or MIC parameters.

On the TEST and CANCEL requests, the TU and MIC parameters specify the location where the system returns the remaining time:

- If you specify TU, the STIMERM macro returns the amount of time remaining to the designated 4-byte storage area. The time is returned as an unsigned 32-bit binary number containing the number of timer units (approximately 26.04166 microseconds per unit) remaining in the interval.

- If you specify MIC, the STIMERM macro returns the remaining time to the designated 8-byte storage area. Bit 51 of the area is the low-order bit of the interval value, and is equivalent to approximately one microsecond.

- If the timer request specified via ID= does not exist for the current task, or has expired, the storage area designated by TU= or MIC= becomes zero. You will receive indication of this.

- If the timer request specified via ID= exists for the current task, and the calculation of the remaining interval results in a negative or zero time interval, the minimum positive interval will be returned to you. This enables you to differentiate the case in which the interval has expired, from the case in which the interval has not yet expired but the remaining interval is less than or equal to zero.

Programming Notes:

- All input and output addresses are treated as full 31-bit addresses.
- The parameter lists may be above or below 16 megabytes.
- There is no interaction between the TTIMER macro support and the STIMERM macro support, or between the STIMER macro support and the STIMERM macro support.
- If the STIMERM macro service cannot access the macro parameter list or any in-storage parameters, the system abends the calling program whether or not it specified an ERRET routine.

The standard form of the STIMERM macro is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede STIMERM.
STIMERM	
b	One or more blanks must follow STIMERM.

	Valid parameters (Required parameters are underlined)
SET	ID,BINTVL, or DINTVL, or GMT, or MICVL, or TOD, or TUINTVL, ERRET, WAIT, EXIT, PARM, RELATED
TEST	ID, TU or MIC, ERRET, RELATED
CANCEL	ID, TU or MIC, ERRET, RELATED
,ID=*stor addr* ,ID=ALL	*stor addr:* A-type address or register (2) - (12). **Note:** ID=ALL is valid only on the CANCEL request.
,TU=*stor addr* ,MIC=*stor addr*	*stor addr:* A-type address or register (2) - (12).
,BINTVL=*stor addr* ,DINTVL=*stor addr* ,GMT=*stor addr* ,MICVL=*stor addr* ,TOD=*stor addr* ,TUINTVL=*stor addr*	*stor addr:* A-type address or register (2) - (12).
,ERRET=*err rtn addr*	*err rtn addr:* A-type address or register (2) - (12).
,WAIT=YES ,WAIT=NO	**Default:** WAIT=NO
,EXIT=*exit rtn addr*	*exit rtn addr:* A-type address or register (2) - (12). **Note:** EXIT must not be specified if WAIT=YES is specified.
,PARM=*stor addr*	*stor addr:* A-type address or register (2) - (12). **Note:** If PARM is specified, EXIT must be specified and WAIT=YES must not be specified.
,RELATED=*value*	

The parameters are explained as follows:

SET
TEST
CANCEL

request to establish, return, or cancel a real time interval. You must specify one of these parameters.

SET indicates a request to establish a real time interval.

TEST indicates a request to return the remaining time for a request made using the SET parameter.

CANCEL indicates a request to cancel and optionally return the remaining time for a timer request.

If the CANCEL parameter specifies (through ID=) a timer request that was established with the WAIT=YES parameter, the task will still remain in the wait condition.

,ID=*stor addr*
,ID=ALL
specifies the address of a 4-byte area containing the identifier assigned to a particular timer request by the timer service routine. ID=ALL is valid only on STIMERM CANCEL; it cancels all the current task's timer requests that were established by the STIMERM SET macro. If you specify ID=ALL, do not specify TU or MIC to request that STIMERM return the remaining time in the interval.

,TU=*stor addr*
,MIC=*stor addr*
specifies that the remaining time in the interval be returned to the 4-byte or 8-byte area specified in *stor addr*. TU or MIC is required for STIMERM TEST and optional for STIMERM CANCEL (providing you do not also specify ID=ALL). TU and MIC are mutually exclusive.

For TU, the time is returned to the specified 4-byte area as an unsigned 32-bit binary number. The low-order bit is approximately 26.04166 microseconds (one timer unit).

For MIC, the time is returned to the specified 8-byte area in microseconds. The 8-byte area stores the remaining interval, which is represented as an unsigned 64-bit binary number; bit 51 is equivalent to 1 microsecond.

,BINTVL=*stor addr*
,DINTVL=*stor addr*
,GMT=*stor addr*
,MICVL=*stor addr*
,TOD=*stor addr*
,TUINTVL=*stor addr*
specify the storage address and format of the time interval to be set. You must specify one of these parameters.

For BINTVL, the address is a 4-byte area containing the time interval. The time interval is represented as an unsigned 32-bit binary number; the low-order bit has a value of one hundredth of a second.

For DINTVL, the address is an 8-byte area in virtual storage containing the time interval. The time interval is represented as zoned decimal digits of the form:

HHMMSSth, where:

HH	is hours (24-hour clock)
MM	is minutes
SS	is seconds
t	is tenths of a second
h	is hundredths of a second

For GMT, the address is an 8-byte area containing the Greenwich mean time at which the interval will complete. The time is represented as zoned decimal digits of the form HHMMSSth, as described previously under DINTVL.

For MICVL, the address is an 8-byte storage area containing the time interval. The time interval is represented as an unsigned 64-bit binary number; bit 51 is the low-order bit of the interval value and equivalent to one microsecond.

For TOD, the address is an 8-byte storage area containing the time of day at which the interval is to be completed. The time of day is represented as zoned decimal digits of the form HHMMSSth, as described previously under DINTVL.

For TUINTVL, the address is a 4-byte area containing the time interval. The time interval is represented as an unsigned 32-bit binary number; the low-order bit has a value of one timer unit (approximately 26.04166 microseconds).

Notes on setting the time interval: For the DINTVL, GMT, and TOD parameters, the zoned decimal digits are not checked for validity. Thus, specifying invalid digits can cause an ABEND 0C7 or an undesired time interval.

The time interval specified by a STIMERM macro has no relation to the time interval specified in an EXEC statement. Specifying a time interval greater than 24 hours with DINTVL, BINVTL, MICVL, or TUINVL, causes the time interval to be set to 24 hours. Specifying a time interval greater than 24 hours with TOD or GMT, causes a 32E abend if ERRET is not specified.

,ERRET=*err rtn addr*

specifies the address of the routine to receive control when the STIMERM function cannot be performed. If you omit this parameter and encounter an error, the invoker of the STIMERM function will be abnormally terminated. The specified error routine will be entered in the addressing mode of the STIMERM invoker. If the macro parameter list or any in-storage parameters are not accessible, the STIMERM invoker will be abended regardless of whether ERRET has been specified.

When the routine receives control, the register contents are:

Register	Contents
0	address of a 24-byte STIMERM parameter list
1	unpredictable
2-14	unchanged
15	return code

,WAIT=*YES*
,WAIT=*NO*

specifies whether the task should be suspended until the requested time interval expires. WAIT=YES specifies that the task should be suspended. WAIT=NO is the default.

You will receive no indication that the timer has expired if:

- You specify WAIT=NO without specifying EXIT
- You do not specify WAIT=NO and the optional exit routine address.

,EXIT = *exit rtn addr*

specifies the address of an exit routine that will gain asynchronous control after the requested timer interval expires. The system's workload and the relative dispatching priority of the associated task determine exactly when, after the interval completes, the exit routine gets control. The specified exit routine will be entered in the addressing mode of the STIMERM invoker. If you specify WAIT = YES, you must *not* specify the EXIT parameter.

Exit Routine Interface

The timer exit routine, established with the EXIT parameter in the STIMERM macro, receives control with the following register values:

R0 - control program information

R1 - points to an 8-byte fetch-protected storage area below 16 megabytes and in the protect key that issued the STIMERM SET macro.

R1----->

Word 1 TIMER REQUEST ID
Word 2 USER PARAMETER (specified in the PARM keyword)

R2-R12 - unpredictable
R13 - address of a 72-byte save area provided by the control program
R14 - return address (to control program)
R15 - address of the exit routine

The exit routine receives control in the addressing mode of the STIMERM issuer. If multiple asynchronous exits are established, the exit routines may not receive control in the same order that the intervals expire.

,PARM = *stor addr*

specifies the address of a 4-byte parameter that the exit routine receives when the requested timer interval expires. You must *not* specify PARM= if you specified WAIT = YES. If you specify PARM=, you must also specify EXIT=.

An exit routine will be unable to distinguish between the case where PARM= was not specified, and the case where the specified PARM value was zero.

,RELATED = *value*

specifies information used to self-document macros by "relating" functions or services to corresponding functions or services. The format and contents of the specified information are at your discretion, and may be any valid macro keyword expression.

Return Codes

When control is returned, register 15 contains one of the following return codes. Note that for non-zero return codes, the ERRET routine receives control (if you specified ERRET). If you did not specify ERRET, a non-zero return code causes the STIMERM invoker to terminate abnormally.

Hexadecimal Code	Meaning
0	The STIMERM service has completed successfully.
8	All time-of-day clocks in the system are inoperative.
10	Parameters passed to STIMERM are invalid.
1C	Request would cause the limit of concurrent STIMERM SET request supported for a task to be exceeded.
24	An invalid STIMERM ID number has been specified. The ID number is either 0, or greater than the highest ID assigned by the system.

Example 1

Operation: Set a timer to a specified time interval. Specify:

- The address of a 4-byte area in which the identifier assigned by the timer service to this request will be returned,
- That control should be given to an asynchronous timer completion exit named TIME when the time interval expires,
- The address of a 4-byte area (containing the time interval represented as an unsigned 32-bit binary number) named INTERVAL. Include an error exit routine named ERROR.

```
STIMERM SET,ID=ADDRESS,BINTVL=INTERVAL,EXIT=TIME,ERRET=ERROR
```

Example 2

Operation: Set a timer to a time interval that specifies the address of a 4-byte area in which the identifier assigned by timer service will be returned. Specify the address of an 8-byte area (containing the Greenwich mean time at which the interval is to be completed) named INTERVAL. Specify that the task should be suspended until the requested time interval expires. Include an error exit routine named EXITX.

```
STIMERM SET,ID=ADDRESS,GMT=INTERVAL,WAIT=YES,ERRET=EXITX
```

Example 3

Operation: Set a timer to a time interval that specifies the address of a 4-byte area in which the identifier assigned by timer service will be returned. Specify the address of an 8-byte area (containing the time interval represented as a zoned decimal digit) in register 8. Specify the address of a 4-byte parameter to be passed to the exit routine when the requested time interval expires. Include the address of an exit error routine in register 9.

```
STIMERM SET,ID=(7),DINTVL=(8),PARM=USERDATA,ERRET=(9)
```

Example 4

Operation: Test the remaining time interval for a timer request established with the SET parameter, specifying the address of a 4-byte area (register 4) from which the identifier assigned to this request by the timer service will be obtained. Specify that the time be returned in a 4-byte area as an unsigned 32-bit binary number at the address labeled INTERVAL. Include the address of an exit error routine called XYZ.

```
STIMERM TEST,ID=(4),TU=INTERVAL,ERRET=XYZ
```

Example 5

Operation: Test the remaining time interval for a timer request established with the SET parameter, specifying the address of a 4-byte area at the address labeled ADDR from which the identifier assigned to this request by timer service will be obtained. Specify that the time be returned in microseconds in an 8-byte area as an unsigned 64-bit binary number at the address labeled INTERVAL. Include the address of an exit error routine called ERRORADD.

```
STIMERM TEST,ID=ADDR,MIC=INTERVAL,ERRET=ERRORADD
```

Example 6

Operation: Cancel a timer request established with a SET parameter, specifying the address of a 4-byte area named ADDRESS containing the identifier assigned by the timer service. The time interval remaining should be returned as an unsigned 32-bit binary number in a 4-byte area called INTERVAL. An exit error routine named ERROR is also specified.

```
STIMERM CANCEL,ID=ADDRESS,TU=INTERVAL,ERRET=ERROR
```

Example 7

Operation: Cancel a timer request established with a SET parameter, specifying the address of a 4-byte area named PLACE containing the identifier assigned by the timer service. The time interval remaining should be returned in an 8-byte area called INTERVAL. An exit error routine named EXITA is also specified.

```
STIMERM CANCEL,ID=PLACE,MIC=INTERVAL,ERRET=EXITA
```

Example 8

Operation: Cancel all the timer requests established with STIMERM SET macro for the current task.

```
STIMERM CANCEL,ID=ALL
```

STIMERM (List Form)

The list form of the STIMERM macro constructs a remote control program parameter list.

The list form of the STIMERM macro is written as follows:

name	***name:*** **symbol. Begin** ***name*** **in column 1.**
b	**One or more blanks must precede STIMERM.**
STIMERM	
b	**One or more blanks must follow STIMERM.**

SET

TEST

CANCEL

,MF=L

,RELATED=*value*

The parameters are explained as follows:

,MF=L
specifies the list form of the STIMERM macro. If you do not specify MF=L, the standard form of the macro is expanded. If you do specify MF=L, the only keyword allowed is RELATED.

Example 1

Operation: Establish a remote STIMERM SET parameter list.

```
STIMERM SET,MF=L
```

Example 2

Operation: Establish a remote STIMERM TEST or CANCEL parameter list.

```
STIMERM TEST,MF=L
```

Example 3

Operation: Establish the appropriate storage for the EXECUTE form of the STIMERM CANCEL macro.

```
STIMERM CANCEL,MF=L
```

STIMERM (Execute Form)

The execute form of the STIMERM macro uses, and can modify, the remote control program parameter list that the list form of the STIMERM macro generates.

The execute form of the STIMERM macro is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede STIMERM.
STIMERM	
b	One or more blanks must follow STIMERM.
	Valid parameters (Required parameters are underlined)
SET	ID,BINTVL, or DINTVL, or GMT, or MICVL, or TOD, or TUINTVL, ERRET, WAIT, EXIT, PARM, RELATED
TEST	ID, TU or MIC, ERRET, RELATED
CANCEL	,ID, TU or MIC, ERRET, RELATED
,ID=ALL	**Note:** ID=ALL is only valid on the CANCEL request.
,ID=*stor addr*	*stor addr:* A-type address or register (2) - (12).
,TU=*stor addr*	*stor addr:* A-type address or register (2) - (12).
,MIC=*stor addr*	*stor addr:* A-type address or register (2) - (12).
,BINTVL=*stor addr* ,DINTVL=*stor addr* ,GMT=*stor addr* ,MICVL=*stor addr* ,TOD=*stor addr* ,TUINTVL=*stor addr*	*stor addr:* A-type address or register (2) - (12).
,ERRET=*err rtn addr*	*err rtn addr:* A-type address or register (2) - (12).
,WAIT=YES ,WAIT=NO	**Default:** WAIT=NO
,EXIT=*exit rtn addr*	*exit rtn addr:* A-type address or register (2) - (12). **Note:** EXIT must not be specified if WAIT=YES is specified.
,PARM=*stor addr*	*stor addr:* A-type address or register (2) - (12). **Note:** If PARM is specified, EXIT must be specified and WAIT=YES must not be specified.
,MF=(E,*ctrl addr*)	*ctrl addr:* A-type address or register (0), (2)-(12) for TEST and CANCEL register (1)-(12) for SET. **Note:** If MF is not specified, the standard form of the macro is expanded.
,RELATED=*value*	

The parameters are explained in the standard form of the STIMERM macro, with the following exception:

,MF=(E,*ctrl addr*)
specifies the execute form of the STIMERM macro using a remote problem program parameter list. If you do not specify MF=(E,ctrl addr), then the standard form of the macro is expanded.

Example 1

Operation: Establish a timer to a specified time interval specifying the address of a 4-byte area in which the identifier assigned to this request by timer service will be returned. Specify:

- The address of an 8-byte area (containing the time interval represented as an unsigned 64-bit binary number) in register 5;
- The address of a program to receive asynchronous control after the requested timer interval expires;
- The address of a 4-byte parameter to be passed to the exit routine when the requested time interval expires.

Include the address of an error routine in register 9.

```
STIMERM SET,ID=(4),MICVL=(5),EXIT=ROUTE,PARM=DATA,                    X
            MF=(E,REMOTE),ERRET=(9)
```

Example 2

Operation: Test the remaining time interval for a timer request established with the SET parameter, specifying the address of a 4-byte area at the address named ADDR in which the identifier assigned by timer service to this request will be returned. Specify that register 3 will point to the appropriate list. Specify that the time be returned in microseconds in an 8-byte area as an unsigned 64-bit binary number at the address named INTERVAL. Include the address of an exit error routine called ERR.

```
STIMERM TEST,ID=ADDR,MIC=INTERVAL,MF=(E,(3)),ERRET=ERR
```

Example 3

Operation: Cancel the timer request established with a SET parameter. Specify the address of a 4-byte identifier named ADDRESS, and that the time interval remaining be returned as an unsigned binary number in a 4-byte area named INTERVAL. Specify an error exit routine named ERROR.

```
STIMERM CANCEL,ID=ADDRESS,TU=INTERVAL,MF=(E,(0)),ERRET=ERROR
```

SYNCH — Take a Synchronous Exit to a Processing Program

The SYNCH macro instruction allows a problem state program to take a synchronous exit to a processing program. On entry to the processing program, the high-order bit, bit 0, of register 14 is set to indicate the addressing mode of the issuer of the SYNCH macro. If bit 0 is 0, the issuer is executing in 24-bit addressing mode; if bit 0 is 1, the issuer is executing in 31-bit addressing mode. The SYNCH routine initializes a PRB (program request block) and schedules execution of the requested program. After the processing program has been executed, the program that issued the SYNCH macro instruction regains control.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction.

The standard form of the SYNCH macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SYNCH.
SYNCH	
b	One or more blanks must follow SYNCH.
entry point addr	*entry point addr:* RX-type address, or register (2) - (12) or (15).
,RESTORE=NO ,RESTORE=YES	**Default:** RESTORE=NO
,AMODE=24 ,AMODE=31 ,AMODE=DEFINED ,AMODE=CALLER	**Default:** AMODE=CALLER. **Note:** AMODE=DEFINED can be specified only if the entry point address is provided in a register.

The parameters are explained as follows:

entry point addr
specifies the address of the entry point of the processing program to receive control.

,RESTORE=NO
,RESTORE=YES
specifies whether registers 2-13 are to be restored when control returns to the caller.

,AMODE=24
,AMODE=31
,AMODE=DEFINED
,AMODE=CALLER
specifies the addressing mode in which the requested program is to receive control.

If AMODE=24 is specified, the requested program will receive control in 24-bit addressing mode.

If AMODE=31 is specified, the requested program will receive control in 31-bit addressing mode.

If AMODE=DEFINED is specified, the user must provide the entry point using a register and not an RX-type address. The requested program will receive control in the addressing mode indicated by the high order bit of the entry point address. If the bit is off, the requested program will receive control in 24-bit addressing mode; if the bit is set, the requested program will receive control in 31-bit addressing mode.

If AMODE=CALLER is specified, the requested program will receive control in the addressing mode of the caller.

Example 1

Operation: Take a synchronous exit to PROGRAMA. Do not restore registers 2-13 when control returns.

```
LOAD    EP=PROGRAMA,DCB=LIB1      Load desired program
LR      R8,R0                     Obtain the entry point
SYNCH   (R8),RESTORE=NO
```

Example 2

Operation: Take a synchronous exit to a program labeled SUBRTN and restore registers 2-13 when control returns.

```
SYNCH   SUBRTN,RESTORE=YES
```

Example 3

Operation: Take a synchronous exit to the program located at the address given in register 8 and restore registers 2-13 when control returns. Indicate that this program is to execute in 24-bit addressing mode.

```
SYNCH (8),RESTORE=YES,AMODE=24
```

Example 4

Operation: Take a synchronous exit to the program located at the address given in register 8 and restore registers 2-13 when control returns. Indicate that this program is to receive control in the addressing mode defined by the high-order bit of its entry point address.

```
SYNCH (8),RESTORE=YES,AMODE=DEFINED
```

Example 5

Operation: Take a synchronous exit to the program located at the address given in register 8 and restore registers 2-13 when control returns. Indicate that this program is to receive control in the addressing mode as the caller.

```
SYNCH (8),RESTORE=YES,AMODE=CALLER
```

SYNCH (List Form)

The list form of the SYNCH macro instruction constructs a control program parameter list.

The list form of the SYNCH macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SYNCH.
SYNCH	
b	One or more blanks must follow SYNCH.

,RESTORE=NO ,RESTORE=YES	**Default:** RESTORE=NO
,AMODE=24 ,AMODE=31 ,AMODE=DEFINED ,AMODE=CALLER	**Default:** AMODE=CALLER
,MF=L	

The parameters are explained under the standard form of the SYNCH macro instruction, with the following exception:

,MF=L
specifies the list form of the SYNCH macro instruction.

Example 1

Operation: Use the list form of the SYNCH macro instruction to specify that registers 2-13 are to be restored when control returns from executing the SYNCH macro instruction and that the addressing mode of the program is to be defined by the high-order bit of the entry point address. Assume that the execute form of the macro instruction specifies the program address.

```
SYNCH ,RESTORE=YES,AMODE=DEFINED,MF=L
```

SYNCH (Execute Form)

The execute form of the SYNCH macro instruction uses a remote control program parameter list that can be generated by the list form of SYNCH.

The execute form of the SYNCH macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede SYNCH.
SYNCH	
b	One or more blanks must follow SYNCH.
entry point addr	*entry point addr:* RX-type address, or register (2) - (12) or (15).
,RESTORE=NO ,RESTORE=YES	**Default:** RESTORE=NO
,AMODE=24 ,AMODE=31 ,AMODE=DEFINED ,AMODE=CALLER	**Default:** AMODE=CALLER **Note:** AMODE=DEFINED can be specified only if the entry point address is provided in a register.
,MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address or register (1), (2) - (12).

The parameters are explained under the standard form of the SYNCH macro instruction, with the following exception:

,MF=(E,*ctrl addr*)
specifies the execute form of the SYNCH macro instruction.

Example 1

Operation: Use the execute form of the SYNCH macro instruction to take a synchronous exit to the program located at the address given in register 8 and restore registers 2-13 when control returns. Indicate that the program is to receive control in the same addressing mode as the caller and that the parameter list is located at SYNCHL2.

```
SYNCH (8),RESTORE=YES,AMODE=CALLER,MF=(E,SYNCHL2)
```

TIME — Provide Time and Date

The TIME macro instruction returns either the local time of day and date, the Greenwich mean time of day and date, or the contents of the TOD clock. The time of day and date are only as accurate as the corresponding information entered by the operator, and the system response time.

Unless STCK is specified, the date is returned in register 1 as packed decimal digits of the form 0CYYDDDF, where:

C	is a digit representing centuries beyond the twentieth. In the years 1900 through 1999, the macro will return a value of C=0.
YY	is the last two digits of the year
DDD	is the day of the year
F	is a 4-bit sign character that allows the data to be unpacked and printed

The time of day, based on a 24-hour clock, is returned in different forms, as designated by the parameters shown below. For the DEC, BIN, and TU parameters, the time of day is returned in register 0. For the MIC and STCK parameters, the time of day and TOD clock contents respectively are stored at the specified address.

The TIME macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede TIME.
TIME	
b	One or more blanks must follow TIME.

DEC BIN TU MIC,*stor addr* STCK,*stor addr*	**Default:** DEC *stor addr:* RX-type address or register (0) or (2) - (12).
,ZONE=LT ,ZONE=GMT	**Default:** ZONE=LT **Note:** This parameter has no meaning if STCK above is specified.
,ERRET=*err rtn addr*	*err rtn addr:* A-type address, or register (2) - (12).

The parameters are explained as follows:

DEC
BIN
TU
MIC ,*stor addr*
STCK ,*stor addr*

specifies that the time of day or TOD clock contents be returned:

For DEC, the time of day is returned in register 0 as packed decimal digits without a sign of the form HHMMSSth, where:

HH	is hours (24-hour clock)
MM	is minutes
SS	is seconds
t	is tenths of seconds
h	is hundredths of seconds

For BIN, the time of day is returned in register 0 as an unsigned 32-bit binary number. The low-order bit is equivalent to 0.01 seconds.

For TU, the time of day is returned in register 0 as an unsigned 32-bit binary number. The low-order bit is approximately 26.04166 microseconds (one timer unit).

For MIC, the time of day is returned in microseconds. The *stor addr* is the address of an 8-byte area in storage with bit 51 equivalent to one microsecond.

For STCK, the contents of the TOD clock is returned as an unsigned 64-bit fixed-point number, where bit 51 is equivalent to 1 microsecond. The *stor addr* is the address of an 8-byte area in storage. Register 1 does not contain the date on return.

Notes:

1. *The resolution of the time-of-day clock is model dependent. See Principles of Operation for an explanation of the rate advancement.*

2. *stor addr must be a 24-bit address.*

,ZONE=LT
,ZONE=GMT
specifies that the local time and date (LT) or the Greenwich mean time and date (GMT) is to be returned.

,ERRET=*err rtn addr*
specifies the address of the routine to be given control when the TIME function cannot be performed because of damaged clocks. The TIME macro will test the return code and give control to the specified routine for a non-zero value. The register contents when the routine is given control are:

Register	Contents
0-1	unpredictable
2-14	unchanged
15	return code

If the caller does not specify ERRET, then the TIME function will return only on successful completion (return code 0). If ERRET is not specified, failure due to damaged clock(s) will result in the abnormal termination of the caller.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion
08	Damaged clocks

Example 1

Operation: Request the system to store the time-of-day clock in the address pointed to by register 2. The user's routine TIMEERR is to receive control if no usable time-of-day clock exists in the system.

```
TIME    STCK,(2),ERRET=TIMEERR
```

TTIMER — Test Interval Timer

The TTIMER macro instruction tests the timer interval established by an STIMER macro instruction. It also cancels the remaining time interval.

If TU is specified or assumed, the TTIMER macro instruction causes the control program to return in register 0 the amount of time remaining in a timer interval previously set by an STIMER macro instruction. The time remaining is returned as an unsigned 32-bit binary number specifying the number of timer units (approximately 26.04166 microsecond units) remaining in the interval. If a time interval has not been set or has already expired, register 0 contains 0.

If MIC is specified, the remaining time is returned to the doubleword area specified in the address. Bit 51 of the area is the low-order bit of the interval value and equivalent to 1 microsecond. If a time interval has not been set or has already expired the area is set to 0.

Note: The resolution of the timer is model dependent. See *Principles of Operation* for additional details concerning the timer facility.

The TTIMER macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede TTIMER.
TTIMER	
b	One or more blanks must follow TTIMER.

CANCEL	
,TU	**Default:** TU
,MIC,*stor addr*	*stor addr:* RX-type address, or register (0) or (2) - (12).
,ERRET=*err rtn addr*	*err rtn addr:* RX-type address, or register (2) - (12).

The parameters are explained as follows:

CANCEL
specifies that the remaining time interval and any exit routine are to be canceled. If the time interval has already expired, the CANCEL option has no effect and a value of zero time remaining is returned. In this case, a specified exit will still receive control. If a non-zero time remaining is returned when the CANCEL option is specified, any exit routine is canceled. If CANCEL is not designated, the unexpired portion of the time interval remains in effect.

If WAIT was coded in the STIMER macro instruction that established the interval, the task is not taken out of the wait condition and CANCEL is ignored.

,TU
,MIC *,stor addr*
specifies that the remaining time in the interval be returned:

For TU, the time is returned in register 0 as an unsigned 32-bit binary number. The low-order bit is approximately 26.04166 microseconds (one timer unit).

For MIC, the time is returned in microseconds. The *stor addr* is the doubleword area on a doubleword boundary where the remaining interval is to be stored.

,ERRET = *err rtn addr*
specifies the address of the routine to be given control when the TTIMER function cannot be performed because of damaged clocks. The TTIMER macro will test the return code and give control to the specified routine for a non-zero value. The register contents when the routine is given control are:

Register	Contents
0-1	unpredictable
2-14	unchanged
15	return code

If the caller does not specify ERRET, then the TTIMER function will return only on successful completion (return code 0). If ERRET is not specified, failure due to damaged clock(s) will result in the abnormal termination of the caller.

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful completion
08	Damaged clocks

Usage Notes:

1. Time intervals established via the STIMERM SET macro instruction cannot be tested or cancelled with the TTIMER macro instruction.

Example 1

Operation: Cancel the task's current time interval. The time remaining, if any, should be returned in timer units in register 0.

```
TTIMER   CANCEL,TU
```

WAIT — Wait for One or More Events

The WAIT macro instruction informs the control program that performance of the active task cannot continue until one or more specific events, each represented by a different ECB (event control block), have occurred. Bit 0 and bit 1 of each ECB must be set to 0 before it is used. The control program takes the following action:

- For each event that has already occurred (each ECB is already posted), the count of the number of events is decreased by 1.
- If the number of events is 0 by the time the last event control block is checked, control is returned to the instruction following the WAIT macro instruction.
- If the number of events is not 0 by the time the last ECB is checked, control is not returned to the issuing program until sufficient ECBs are posted to bring the number to 0. Control is then returned to the instruction following the WAIT macro instruction.

The WAIT macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WAIT.
WAIT	
b	One or more blanks must follow WAIT.
event nmbr,	*event nmbr:* symbol, decimal digit, or register (0) or (2) - (12). **Default:** 1 **Value range:** 0-255
ECB=*ecb addr* ECBLIST=*ecb list addr*	*ecb addr:* RX-type address, or register (1) or (2) - (12). *ecb list addr:* RX-type address, or register (1) or (2) - (12).
,LONG=NO ,LONG=YES	**Default:** LONG=NO
,RELATED=*value*	*value:* Any valid macro keyword specification.

The parameters are explained as follows:

event nmbr,
specifies the number of events waiting to occur.

ECB=*ecb addr*
ECBLIST=*ecb list addr*
specifies the address of an ECB on a fullword boundary or the address of a virtual storage area containing one or more consecutive fullwords on a fullword boundary. Each fullword contains the address of an ECB; the high order bit in the last fullword must be set to 1 to indicate the end of the list.

The ECB parameter is valid only if the number of events is specified as one or is omitted. The number of ECBs in the list specified by the ECBLIST form must be equal to or greater than the specified number of events.

,LONG=NO
,LONG=YES
specifies whether the task is entering a long wait (YES) or a regular wait (NO).

,RELATED=*value*
specifies information used to self-document macro instructions by "relating" functions or services to corresponding functions or services. The format and contents of the information specified are at the discretion of the user, and may be any valid coding values.

The RELATED parameter is available on macro instructions that provide opposite services (for example, ATTACH/DETACH, GETMAIN/FREEMAIN, and LOAD/DELETE), and on macro instructions that relate to previous occurrences of the same macro instructions (for example, CHAP and ESTAE).

The RELATED parameter may be used, for example, as follows:

```
WAIT1    WAIT      1,ECB=ECB,RELATED=(RESUME1,
                   'WAIT FOR EVENT')
   .
   .
   .
RESUME1  POST      ECB,0,RELATED=(WAIT1,
                   'RESUME WAITER')
```

Note: Each of these macro instructions will fit on one line when coded, so there is no need for a continuation indicator.

CAUTION
A job step with all of its tasks in a WAIT condition is terminated upon expiration of the time limits that apply to it.

Example: You have previously initiated one or more activities to be completed asynchronously to your processing. As each activity was initiated, you set up an ECB in which bits 0 and 1 were set to 0. You now wish to suspend your task via the WAIT macro instruction until a specified number of these activities have been completed.

Completion of each activity must be made known to the system via the POST macro instruction. POST causes an addressed ECB to be marked complete. If completion of the event satisfies the requirements of an outstanding WAIT, the waiting task is marked ready and will be executed when its priority allows.

Example 1

Operation: Wait for one event to occur (with a default count).

```
        WAIT  ECB=WAITECB
WAITECB DC    F'0'
```

Example 2

Operation: Wait for 2 events to occur.

```
          WAIT   2,ECBLIST=LISTECBS
          .
          .
LISTECBS  DC  A(ECB1)
          DC  A(ECB2)
          DC  X'80'
          DC  AL3(ECB3)
```

Example 3

Operation: Enter a long wait for a task.

```
          WAIT   1,ECBLIST=LISTECBS,LONG=YES
          .
          .
          .
LISTECBS  DC  A(ECB1)
          DC  A(ECB2)
          DC  X'80'
          DC  AL3(ECB3)
```

WTL — Write To Log

The WTL macro instruction writes a message to the system log. The message can include any character that can be used in a C-type (character) DC statement, and is assembled as a variable-length record.

Note: The exact format of the output of the WTL macro instruction varies depending on the job entry subsystem (JES2 or JES3) that is being used, the output class that is assigned to the log at system initialization, and whether DLOG is in effect for JES3. In JES3, system log entries are preceded by a 23-character prefix that includes a time stamp and routing information. If the combined prefix and message exceeds 126 characters, the log entry is split at the first blank or comma encountered when scanning backward from the 126th character of the combined prefix and message. See *Operations: JES3 Commands* for information about the format of the log entry when using JES3.

The standard form of the WTL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTL.
WTL	
b	One or more blanks must follow WTL.
'msg'	*msg:* Up to 126 characters.

The parameter is explained as follows:

'msg'
specifies the message to be written to the system log. The message must be enclosed in apostrophes, which will not appear in the system log. See Figure 43 on page 109 for a list of the printable EBCDIC characters passed to display devices or printers.

Note: If the msg text exceeds 126 characters, truncation occurs at the last embedded blank before the 126th character; when there are no embedded blanks, truncation occurs after the 126th character.

Example 1

Operation: Write a message to the system log.

```
WTL   'THIS IS THE STANDARD FORMAT FOR THE WTL MACRO'
```

Example 2

Operation: Write a message constructed in the list form of WTL.

```
WTL   MF=(E,(R2))
```

WTL (List Form)

The list form of the WTL macro instruction is used to construct a control program parameter list. The message parameter must be provided in the list form of the macro instruction.

The list form of the WTL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTL.
WTL	
b	One or more blanks must follow WTL.
'msg'	*msg:* Up to 126 characters.
,MF=L	

The *'msg'* parameter is explained under the standard form of the WTL macro instruction. A description of the MF parameter follows:

,MF=L
specifies the list form of the WTL macro instruction.

Note: If msg text exceeds 126 characters, truncation occurs at the last embedded blank before the 126th character; when there are no embedded blanks, truncation occurs after the 126th character.

WTL (Execute Form)

The execute form of the WTL macro instruction uses a remote control program parameter list. The parameter list can be generated by the list form of WTL. You cannot modify the message in the execute form.

The execute form of the WTL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTL.
WTL	
b	One or more blanks must follow WTL.
MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) or (2) - (12).

This parameter is explained as follows:

MF=(E,*ctrl addr*)
specifies the execute form of the WTL macro instruction. This form uses a remote control program parameter list.

WTO — Write to Operator

The WTO macro instruction writes a message to one or more operator consoles. WTO processing uses register 15.

The standard form of the WTO macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTO.
WTO	
b	One or more blanks must follow WTO.

'msg' *('text')* *('text',line type)*	*msg:* Up to 125 characters. The permissible *line types* and lengths are shown below: C 34 char L 70 char D 70 char DE 70 char E --- **Default:** D The maximum number of each line type allowed in a single WTO instructions is: 1 C type 2 L type 10 D type 1 DE type 1 E type The maximum total number of line types allowed in one instruction is 10.
,ROUTCDE=(*routing code*)	*routing code:* decimal digit from 1 to 16. The *routing code* is one or more codes, separated by commas.
,DESC=(*desc code*)	*desc code:* decimal digit from 1 to 16. The *desc code* is one or more codes, separated by commas.

The parameters are explained as follows:

'msg'
('text')
('text',line type)
specifies the message or multiple-line message to be written to one or more operator consoles.

The first format is used to write a single-line message to the operator. In the format, the message must be enclosed in apostrophes, which do not appear on the console. It can include any character that can be used in a character (C-type) DC instruction. When a program issues a WTO macro instruction, the control program translates the text; only standard printable EBCDIC characters are passed to the display devices as shown in Figure 43 on page 109. All other characters are replaced by blanks. Unless the console has dual-case capability, lowercase characters are converted to uppercase by the display station or printer and displayed or printed as uppercase characters. The message is assembled as a variable-length record.

The second and third formats are used to write a multiple-line message to the operator. The message can be up to ten lines long; the system truncates the message at the end of the tenth line. The ten-line limit does not include the control line (message IEE9321I), as explained under line type C below.

Note: If the second format is coded without repetition, for example, (*'text'*), the message appears as a single-line message.

The text is one line of the multiple-line message. A line consists of a character string enclosed in apostrophes (which do not appear on the operator console). Any character valid in a C-type DC instruction can be coded. The maximum number of characters depends on which line type is specified.

Note: The left most three bytes of register zero must be zero for a multiple-line message. The user must ensure that this is done.

The line type defines the type of information contained in the "text" field of each line of the message:

C

indicates that the "text" parameter is the text to be contained in the control line of the message. The control line normally contains a message title. C may only be coded for the first line of a multiple-line message. If this parameter is omitted and descriptor code 9 is coded, the system generates a control line (message IEE932I) containing only a message identification number. The control line remains static during framing operations on a display console (provided that the message is displayed in an out-of-line display area). Control lines are optional.

L

indicates that the "text" parameter is a label line. Label lines contain message heading information; they remain static during framing operations on a display console (provided that the message is displayed in an out-of-line display area). Label lines are optional. If coded, lines must either immediately follow the control line or another label line or be the first line of the multiple-line message if there is no control line. Only two label lines may be coded per message.

D

indicates that the "text" parameter contains the information to be conveyed to the operator by the multiple-line message. During framing operations on a display console, the data lines are paged.

DE

indicates that the "text" parameter contains the last line of information to be passed to the operator.

E

indicates that the previous line of text was the last line of text to be passed to the operator. The "text" parameter, if any, coded with a line type of E is ignored.

,ROUTCDE = *routing code*

specifies the routing code(s) to be assigned to the message.

The routing codes are:

1	Master console action	9	System security
2	Master console information	10	System error/maintenance
3	Tape pool	11	Programmer information
4	Direct access pool	12	Emulators
5	Tape library	13	Reserved for customer use
6	Disk library	14	Reserved for customer use
7	Unit record pool	15	Reserved for customer use
8	Teleprocessing control	16	Reserved for future expansion

Note: Routing codes 1, 2, 3, 4, 7, 8, and 10 cause hard copy of the message when display consoles are used or more than one console is active. All other routing codes may go to hard copy as a SYSGEN option or as a result of a VARY HARDCPY command.

,DESC = (*desc code***)**

specifies the message descriptor code(s) to be assigned to the message. Descriptor codes 1 through 6 and descriptor code 11 are mutually exclusive. Codes 7 through 10 can be assigned in combination with any other code.

The descriptor codes are:

1	System failure	7	Application program/processor
2	Immediate action required	8	Out-of-line message
3	Eventual action required	9	Operator request
4	System status	10	Dynamic status displays
5	Immediate command response	11	Critical eventual action requested
6	Job status	12-16	Reserved for future use

Note: All WTO messages with descriptor codes of 1, 2, or 11 are action messages that have an @ sign printed before the first character. This indicates a need for operator action.

Messages with descriptor code 7 are deleted at end of job step.

Support for queuing messages with descriptor code 8 is by console id only.

On operator consoles that support color, descriptor codes determine the color in which a message should be displayed. The colors used are described in *Operations: System Commands.*

The message processing facility cannot suppress messages with descriptor code 5 (except for responses from MONITOR JOBNAMES, MONITOR SESS, and MONITOR STATUS commands). Messages with any other descriptor codes can be suppressed if they have been identified by an id or prefix in SYS1.PARMLIB member MPFLSTxx.

When control is returned, general register 1 contains the identification number (24 bits and right-justified) assigned to the message. This number can be used to delete the message when it is no longer needed.

Return codes from execution of a WTO using the multiple-line feature are as follows:

Hexadecimal Code	Meaning
00	No errors encountered.
04	Number of lines passed was 0; request is ignored. Number of lines passed was greater than 10; only 10 lines are processed. Message text length for a line was less than 1; all lines up to error line are processed.
08	ID passed in register 0 does not match any on queue. Request is ignored.
0C	Invalid line type. An end has been forced at the point of the error except if the first line is an E line, in which case the request is ignored.

Return codes from execution of a WTO are as follows:

Hexadecimal Code	Meaning
30	Required resource for routing code 11 was not available. Request is ignored for routing code 11. If any other routing code is specified, the request is processed.

Example 1

Operation: Write a WTO message to all active consoles.

```
WTO   'NDP00005   ENDED',                                    X
      ROUTCDE=(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16), X
      DESC=(4)
```

Example 2

Operation: Write a multiple-line message to the master console if the master is receiving routing code 2 and to any other console receiving routing code 2.

```
WTO      ('text 1',D),     DATA LINE                    X
         ('text 2',DE),    DATA END LINE                X
         ROUTCDE=(2),DESC=(4)
```

WTO (List Form)

The list form of the WTO macro instruction is used to construct a control program parameter list.

The list form of the WTO macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTO.
WTO	
b	One or more blanks must follow WTO.
'msg' (*'text'*)	*msg:* Up to 125 characters. The permissible *line types* and *text* lengths are shown below: C 34 char L 70 char D 70 char DE 70 char E --- **Default:** D The maximum number of each line type allowed in a single WTO instructions is: 1 C type 2 L type 10 D type 1 DE type 1 E type. The maximum total number of line types allowed in one instruction is 10.
,ROUTCDE=(*routing code*)	*routing code:* decimal digit from 1 to 16. The *routing code* is one or more codes, separated by commas.
,DESC=(*desc code*)	*desc code:* decimal digit from 1 to 16. The *desc code* is one or more codes, separated by commas.
,MF=L	

The parameters are explained under the standard form of the WTO macro instruction, with the following exception:

,MF=L
specifies the list form of the WTO macro instruction.

WTO (Execute Form)

The execute form of the WTO macro instruction uses a remote control program parameter list. The parameter list can be generated by the list form of WTO. The message cannot be modified in the execute form of the macro instruction.

The execute form of the WTO macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTO.
WTO	
b	One or more blanks must follow WTO.
MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) or (2) - (12).

This parameter is explained as follows:

MF=(E,*ctrl addr*)
specifies the execute form of the WTO macro instruction using a remote control program parameter list.

Example 1

Operation: Write a message with a pre-built parameter list pointed to by register 1.

```
WTO     MF=(E,(1))
```

WTOR — Write to Operator with Reply

The WTOR macro instruction writes a message requiring a reply to one or more operator consoles and the hardcopy log. The macro instruction also provides the information required by the control program to return the reply to the issuing program.

This macro can be assembled compatibly between MVS/XA and MVS/370 through the use of the SPLEVEL macro instruction. Default processing will result in an expansion of the macro that operates only with MVS/XA. See the topic "Selecting the Macro Level" for additional information. If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction.

The standard form of the WTOR macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTOR.
WTOR	
b	One or more blanks must follow WTOR.
'msg'	*msg:* Up to 122 characters.
,reply addr	*reply addr:* A-type address, or register (2) - (12).
,reply length	*reply length:* symbol, decimal digit, or register (2) - (12). The minimum length is 1; the maximum length is 115 when the operator enters REPLY id, 'reply' and 119 when the operator enters R id, 'reply'.
,ecb addr	*ecb addr:* A-type address, or register (2) - (12).
,ROUTCDE=(*routing code*)	*routing code:* decimal digit from 1 to 16. The *routing code* is one or more codes, separated by commas.

The parameters are explained as follows:

'msg'
specifies the message to be written to the operator's console. The message must be enclosed in apostrophes, which do not appear on the console. It can include any character that can be used in a character (C-type) DC instruction. When a program issues a WTOR macro instruction, the control program translates the text; only the standard printable EBCDIC characters shown in Figure 43 on page 109 are passed to the display devices. All other characters are replaced by blanks. Unless the console has dual-case capability, lowercase characters are converted to uppercase by the display station or printer and displayed or printed as uppercase characters. The message is assembled as a variable-length record.

Note: All WTOR messages are action messages. An indicator is printed before the first character of an action message to indicate a need for operator action.

,reply addr
specifies the address in virtual storage of the area into which the control program is to place the reply. The reply is left-justified at this address.

,reply length
specifies the length, in bytes, of the reply message.

,ecb addr

specifies the address of the event control block (ECB) to be used by the control program to indicate the completion of the reply and the id of the replying console. After the control program receives the reply, the ECB appears as follows:

Offset	Length(bytes)	Contents
0	1	Completion code
1	2	Reserved
3	1	Console id in hexadecimal

,ROUTCDE=(*routing code*)

specifies the routing code(s) to be assigned to the message.

The routing codes are:

1	Master console action	9	System security
2	Master console information	10	System error/maintenance
3	Tape pool	11	Programmer information
4	Direct access pool	12	Emulators
5	Tape library	13	Reserved for customer use
6	Disk library	14	Reserved for customer use
7	Unit record pool	15	Reserved for customer use
8	Teleprocessing control	16	Reserved for future expansion

When control is returned, general register 1 contains the identification number (24 bits and right-justified) assigned to the message. This number can be used to delete the message when a reply is no longer needed.

Example 1

Operation: Write a WTOR message to all active consoles.

```
WTOR   'THIS IS WTOR NUMBER 001',REPLY,18,ECB1,                X
       ROUTCDE=(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)
```

WTOR (List Form)

The list form of the WTOR macro instruction is used to construct a control program parameter list. The message parameter must be provided in the list form.

The list form of the WTOR macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTOR.
WTOR	
b	One or more blanks must follow WTOR.
'msg'	*msg:* Up to 122 characters.
, *,reply addr*	*reply addr:* A-type address.
, *,reply length*	*reply length:* symbol or decimal digit. The minimum length is 1; the maximum length is 115 when the operator enters REPLY id, 'reply' and 119 when the operator enters R id, 'reply'.
, *,ecb addr*	*ecb addr:* A-type address.
,ROUTCDE=(*routing code*)	*routing code:* decimal digit from 1 to 16. The *routing code* is one or more codes, separated by commas.
,MF=L	

The parameters are explained under the standard form of the WTOR macro instruction, with the following exception:

,MF=L
specifies the list form of the WTOR macro instruction.

WTOR (Execute Form)

The execute form of the WTOR macro instruction uses a remote control program parameter list. The parameter list can be generated by the list form of WTOR.

The execute form of the WTOR macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WTOR.
WTOR	
b	One or more blanks must follow WTOR.
, ,*reply addr*	*reply addr:* RX-type address, or register (2) - (12).
, ,*reply length*	*reply length:* symbol, decimal digit, or register (2) - (12). The minimum length is 1; the maximum length is 115 when the operator enters REPLY id, 'reply' and 119 when the operator enters R id, 'reply'.
, ,*ecb addr*	*ecb addr:* RX-type address, or register (2) - (12).
,MF=(E,*ctrl addr*)	*ctrl addr:* RX-type address, or register (1) or (2) - (12).

The parameters are explained under the standard form of the WTOR macro instruction, with the following exception:

,MF=(E,*ctrl addr*)
specifies the execute form of the WTOR macro instruction using a remote control program parameter list. The parameter list must be aligned on a fullword boundary. The list form of WTOR provides this alignment.

XCTL — Pass Control to a Program in Another Load Module

The XCTL macro instruction passes control to a specified entry name in another load module; the entry name must be a member name, an alias in a directory of a partitioned data set, or must have been specified in an IDENTIFY macro instruction. The control program brings the load module containing the entry name into storage if a usable copy is not already available. XCTL handles the setting of the addressing mode when passing control to this entry name. The control program reassigns the storage occupied by the load module that issued the XCTL if that module is no longer required. The program executing the XCTL macro instruction is logically removed as a subprogram of the program (system or user) that placed the issuer of XCTL into execution.

No return is made to the program issuing the XCTL macro instruction; the use count for the load module containing the XCTL macro instruction is decremented by 1. A return to the program that placed the issuer of XCTL into execution is required for successful completion of the task. For this reason, registers 2 through 14, the program interruption control area, and the program mask must be restored to the state that existed when the load module received control before the XCTL macro instruction can be issued. If the specified entry cannot be located, the task is abnormally terminated.

On entry to the program specified in the XCTL macro, the high-order bit, bit 0, of register 14 is set to indicate the addressing mode of the issuer of the macro. If bit 0 is 0, the issuer is executing in 24-bit addressing mode; if bit 0 is 1, the issuer is executing in 31-bit addressing mode.

If your program is to execute in 31-bit addressing mode, you must use the MVS/XA version of this macro instruction.

The standard form of the XCTL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede XCTL.
XCTL	
b	One or more blanks must follow XCTL.

(*reg1*), (*reg1*,*reg2*),	*reg1 and reg2:* decimal digits in the order 2 through 12.
EP=*entry name* EPLOC=*entry name addr* DE=*list entry addr*	*entry name:* symbol. *entry name addr:* A-type address or register (2) - (12). *list entry addr:* A-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* A-type address, or register (2) - (12).
,LSEARCH=NO ,LSEARCH=YES	**Default:** LSEARCH=NO

The parameters are explained as follows:

(reg1),
(reg1,reg2),
specifies the register or range of registers to be restored from the save area at the address contained in register 13.

EP = *entry name*
EPLOC = *entry name addr*
DE = *list entry addr*

specifies the entry name, the address of the entry name, or the address of a 60-byte list entry for the entry name that was constructed using the BLDL macro instruction. If EPLOC is coded, the name must be padded to eight bytes, if necessary.

If you issue the XCTL macro instruction and the DE parameter specifies an entry in an authorized library, the program-supplied DE information is ignored for integrity reasons. Instead, contents management uses the BLDL macro instruction to construct a new list entry containing the DE information for the XCTL. The DE information supplied by an unauthorized program will also be ignored if the XCTL macro instruction is requesting access to a program or library that is controlled by the System Authorization Facility.

Note: When you use the DE parameter with the XCTL macro, DE specifies the address of a list that was created by a BLDL macro. BLDL and XCTL must be issued from the same task; otherwise, the system might terminate the program with an abend code of 106 and a return code of 15. Therefore, do not issue an ATTACH or a DETACH macro between issuances of the BLDL and XCTL macros.

,DCB = *dcb addr*

specifies the address of the opened data control block for the partitioned data set containing the entry name described above. This parameter must indicate the same DCB used in the BLDL mentioned above. The DCB must not be defined in the program issuing the XCTL macro instruction.

If the DCB parameter is omitted or if DCB=0 is specified when the XCTL macro instruction is issued by the job step task, the data sets referred to by either the STEPLIB or JOBLIB DD statement are first searched for the entry name. If the entry name is not found, the link library is searched.

If the DCB parameter is omitted or if DCB=0 is specified when the XCTL macro instruction is issued by a subtask, the data sets associated with one or more data control blocks referred to by the TASKLIB operand of previous ATTACH macro instructions in the subtasking chain are first searched for the entry point name. If the entry point name is not found, the search is continued as if the XCTL had been issued by the job step task.

Note: DCB must reside in 24-bit addressable storage.

,LSEARCH=NO
,LSEARCH=YES

specifies whether (YES) or not (NO) you want the search limited to the job pack area and the first library in the normal search sequence.

Note: Do not use register 1 as a pointer to the parameter list passed by the module that issues XCTL. Use the execute form of the XCTL and pass the parameters explicitly using the PARAM keyword.

Example 1

Operation: Pass control via the address of the entry name (XCTLEP), and have registers 2-12 restored. Let the system determine the copy of the module to be used.

```
XCTL    (2,12),EPLOC=XCTLEP
```

XCTL (List Form)

Two parameter lists are used in an XCTL macro instruction: a control program parameter list and an optional problem program parameter list. Only the control program parameter list can be constructed in the list form of XCTL. Address parameters to be passed in a parameter list to the problem program can be provided using the list form of the CALL macro instruction. This parameter list can be referred to in the execute form of XCTL.

The list form of the XCTL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede XCTL.
XCTL	
b	One or more blanks must follow XCTL.

EP=*entry name*, EPLOC=*entry name addr*, DE=*list entry addr*,	*entry name:* symbol. *entry name addr:* A-type addresses. *list entry addr:* A-type address.
DCB=*dcb addr*,	*dcb addr:* A-type address.
LSEARCH=NO, LSEARCH=YES,	**Default:** LSEARCH=NO
SF=L	

The parameters are explained under the standard form of the XCTL macro instruction, with the following exception:

SF=L
specifies the list form of the XCTL macro instruction.

Note: Coding the LSEARCH parameter causes a parameter list to be created that is different from the list created when LSEARCH is omitted. If you code LSEARCH in either the list or execute form of the macro instruction, you must code it in both forms.

XCTL (Execute Form)

Two parameter lists are used in the XCTL macro instruction: a control program parameter list and problem program parameter list. Either or both of these parameter lists can be remote and can be referred to, and modified by, the execute form of XCTL. If only the problem program parameter list is remote, parameters that require the control program parameter list cause that list to be constructed inline as part of the macro expansion. If only the control program parameter list is remote, no problem program parameters can be specified.

The execute form of the XCTL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede XCTL.
XCTL	
b	One or more blanks must follow XCTL.

(*reg1*), (*reg1*,*reg2*),	*reg1* and *reg2:* decimal digits or RX-type addresses, and in the order 2 through 12.
EP=*entry name*, EPLOC=*entry name addr*, DE=*list entry addr*,	*entry name:* symbol. *entry name addr:* RX-type address of register (2) - (12). *list entry addr:* RX-type address, or register (2) - (12).
DCB=*dcb addr*,	*dcb addr:* RX-type address, or register (2) - (12).
PARAM=(*addr*), PARAM=(*addr*),VL=1,	*addr:* RX-type address, or register (2) - (12). *addr* is one or more addresses, separated by commas. For example, PARAM=(*addr*,*addr*,*addr*)
LSEARCH=NO, LSEARCH=YES,	**Default:** LSEARCH=NO
MF=(E,*prob addr*) SF=(E,*ctrl addr*) MF=(E,*prob addr*),SF=(E,*ctrl addr*)	*prob addr:* RX-type address, or register (1) or (2) - (12). *ctrl addr:* RX-type address, or register (2) - (12) or (15).

The parameters are explained under the standard form of the XCTL macro instruction, with the following exceptions:

PARAM=(*addr*)
PARAM=(*addr*),VL=1

specifies address(es) to be passed to the called program. Each address is expanded inline to a fullword on a fullword boundary, in the order designated. Register 1 contains the address of the first parameter when the program is given control. If this parameter is not coded, register 1 is not altered unless the LSEARCH parameter is coded. If LSEARCH is coded, the contents of register 1 are unpredictable.

VL=1 should be designated only if the called program can be passed a variable number of parameters. VL=1 causes the high-order bit of the last address parameter to be set to 1; the bit can be checked to find the end of the list.

LSEARCH=NO
LSEARCH=YES

specifies whether (YES) or not (NO) you want the search limited to the job pack area and to the first library in the normal search sequence. If LSEARCH is specified and PARAM is not specified, the contents of register 1 are unpredictable.

MF=(E,*prob addr*)
SF=(E,*ctrl addr*)
MF=(E,*prob addr*),SF=(E,*ctrl addr*)
specifies the execute form of the XCTL macro instruction. This form uses a remote problem program parameter list, a remote control program parameter list, or both.

Note: Coding the LSEARCH parameter causes a parameter list to be created that is different from the list created when LSEARCH is omitted. If you code LSEARCH in either the list or execute form of the macro instruction, you must code it in both forms.

Index

Special Characters

A

D

E

F

G

H

I

J

N

O

P

Q

R

S

X

Numerics

MVS/Extended Architecture Supervisor Services and Macro Instructions

GC28-1154-4

S370-36

Printed in U.S.A.

MVS/Extended Architecture
Supervisor Services and
Macro Instructions

READER'S
COMMENT
FORM

GC28-1154-4

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 Supervisor Services and Macro Instructions

GC28-1154-4 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®

GC28-1154-04