GC28-0683-1
File No. S370-36

Systems

OS/VS2 Supervisor Services and Macro Instructions

VS2 Release 3

p 21, 24, 114

? 30, 55, 57, 93

IBM

Second Edition (June, 1975)

This is a reprint of GC28-0683-0 incorporating changes released in the following Technical Newsletter:

GN28-2589 (dated February 28, 1975)

This edition applies to release 3 of OS/VS2 and to all subsequent VS2 releases until otherwise indicated in new editions or Technical Newsletters. Changes are continually made 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.

Requests for copies of 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, Publications Development, Department D58, Building 706-2, PO Box 390, Poughkeepsie, N.Y. 12602. Comments become the property of IBM.

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 *OS/VS2 System Programming Library: Job Management, Supervisor, and TSO,* GC28-0682.

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 eight topics. Specific topics include:

Linkage Conventions: Well designed programs use system resources efficiently. Knowing the conventions and characteristics of the supervisor will help you design more efficient programs.

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: The supervisor can be used to aid communication between segments of a program. Save area, addressability, and passage of control from one segment of a program to another are discussed.

Resource Control: Portions of some tasks depend on the completion of events in other tasks. This requires planned task synchronization. Planning is also required when more than one program uses a serially reusable resource.

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.

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.

Miscellaneous Services: In addition to the services outlined above, facilities are provided for timing events, extended precision floating-point simulation, and operator communication with both the system and application programs.

Part II contains the descriptions and definitions of the supervisor macro instructions available in the OS/VS 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.

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

OS/VS2 Planning Guide for Release 2, GC28-0667

OS/VS - DOS/VS - VM/370 Assembler Language, GC33-4010

OS/VS
Checkpoint/Restart, GC26-3784
Data Management Macro Instructions, GC26-3793
Data Management Services Guide, GC26-3783
Linkage Editor and Loader, GC28-6451
Services Aids, GC28-0633

IBM System/370
Principles of Operation, GA22-7000

OS/VS2 System Programming Library: Job Management, Supervisor, and TSO, GC28-0682

Contents

Illustrations

Figures

Summary of Amendments for GC28-0683-0 as Updated by GN28-2589 VS2 Release 3

Part I: Supervisor Services

The basic changes in this part of the manual are found in the chapters:

Resource Control
(EVENTS)

Interruption, Termination, and Dumping Services
(Interface to an ESTAE exit)

Part II: Macro Instructions

The new material in this part of the manual is the insertion of the EVENTS macro instruction.
The basic changes in this part of the manual are found in the Introduction to Supervisor Macro Instructions (Continuation Lines) section and the following macro instructions and parameters:

DEQ (List Form)

IDENTIFY
Add an Entry Name

POST
Signal Event Completion

STATUS
Change Subtask Status

WTO
Write to Operator

WTOR (Execute Form)

Summary of Amendments for GC28-0683-0 VS2 Release 2

Information in this manual applies to VS2 only.

Part I: Supervisor Services

The basic changes in this part of the manual are found in the chapters:

Interruption, Termination, and Dumping Services
(ABEND, ESTAE, and SETRP routines)

Real Storage Management
(PGLOAD and PGOUT routines)

Part II: Macro Instructions

The basic changes in this part of the manual are found in the following macro instructions and parameters:

ABEND
- SYSTEM and USER to designate system or user completion code.
- DUMPOPT= to produce a tailored dump (via using SNAP).

ATTACH
- ESTAI= to specify an ESTAI exit.
- TERM= to schedule exit routine for additional situations.
- RELATED= to self-document macro instruction.

CHAP
- RELATED= to self-document macro instruction.

DELETE
- RELATED= to self-document macro instruction.

DEQ
- RELATED= to self-document macro instruction.

DETACH
- RELATED= to self-document macro instruction.

DOM
- REPLY= to eliminate need for a reply to a WTOR.

ENQ
- RELATED= to self-document macro instruction.

ESTAE
- to extend recovery capabilities of STAE.

FREEMAIN
- LC, LU, VC, VU, EC, EU, RC, and RU to maintain compatibility with GETMAIN.
- RELATED= to self-document macro instruction.

GETMAIN
- RC and RU to maintain compatibility with FREEMAIN.
- RELATED= to self-document macro instruction.

LINK
- ERRET= to schedule error routine.

LOAD
- ERRET= to schedule error routine.
- RELATED= to self-document macro instruction.

PGLOAD
- to load virtual areas into real areas.

PGOUT
- to page out virtual areas from real areas.

SETRP
- to indicate requests a recovery exit may make.

SNAP
- SDATA=(LSQA, SQA, SWA) to allow more areas to be dumped.

STATUS
- SYNCH to stop all subtasks of the caller.

STIMER
- GMT= to return Greenwich mean time.
- ERRET= to schedule error routine.

TIME
- STCK to return TOD clock as unsigned 64-bit fixed-point number.
- ZONE= to return local or Greenwich mean time and date.
- ERRET= to schedule error routine.

TTIMER
- ERRET= to schedule error routine.

WAIT
- LONG= to specify a long wait.

Part I: Supervisor Services

Summary of Services

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

The services you can request from the supervisor can be classified as follows:

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: The supervisor can be used to aid communication between segments of a program. Save areas, addressability, and passage of control from one segment of a program to another are 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.

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.

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.

In addition to the services outlined above, the supervisor provides the facilities for timing events, extended precision floating-point simulation, and operator communication with both the system and application programs.

All programs, regardless of function or relative position in the task, should be designed using certain conventions and with certain characteristics of the control program in mind. This chapter describes these conventions and characteristics and discusses the effects they may have on the execution of your program.

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. The low-order three bytes 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 immedately, you should load the address from register 1 into one of registers 2-12 or store the address in a fullword in your program.

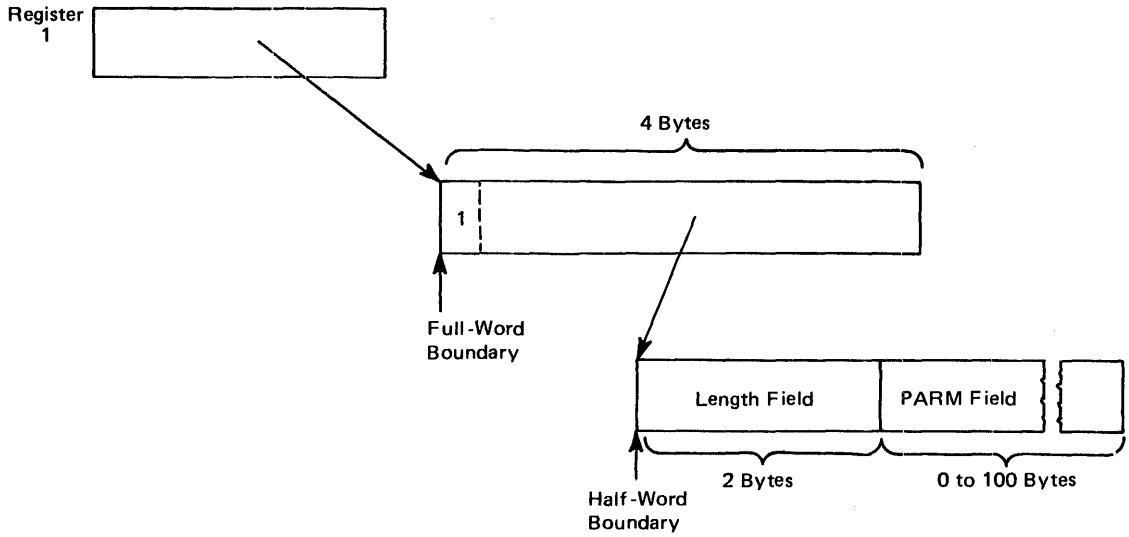


Figure 1. Acquiring PARM Field Information

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
1	Used by PL/I language program
2	Address of previous save area (stored by calling program)
3	Address of next save area (stored by current program)
4	Register 14 (Return address)
5	Register 15 (Entry address)
6	Register 0
7	Register 1
8	Register 2
9	Register 3
10	Register 4
11	Register 5
12	Register 6
13	Register 7
14	Register 8
15	Register 9
16	Register 10
17	Register 11
18	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. SAVE Macro Instruction Used to Save (A) all Registers but 13 and (B) Registers 5-10, 14 and 15

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

Establishing a Base Register

In System/370, 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, a base register can be established 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.

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, the address of the calling program's save area, which you used, must be saved, 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, the following procedure should be followed:

- 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.

Figures 4 and 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.

```
PROGRAM     CSECT
            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(3)
```

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
            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 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 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 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 the low-order byte of register 15 if one is required. Otherwise, restore register 15.

One task is created in the address space by the control program 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 central processing unit. 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 hexadecimal code of '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.

Caution: All modules contained in the libraries for a job step should be uniquely named. If duplicate module names are contained in these libraries, the results are unpredictable.

Priorities

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

Address Space Priority

Each job initiated results in the creation of an address space. 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 central processing unit time and other system resources.

It may be desireable 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.

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 since 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 since more than one task may be executing simultaneously in the same address space on different central processing units. 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 or a subtask is changed. 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.

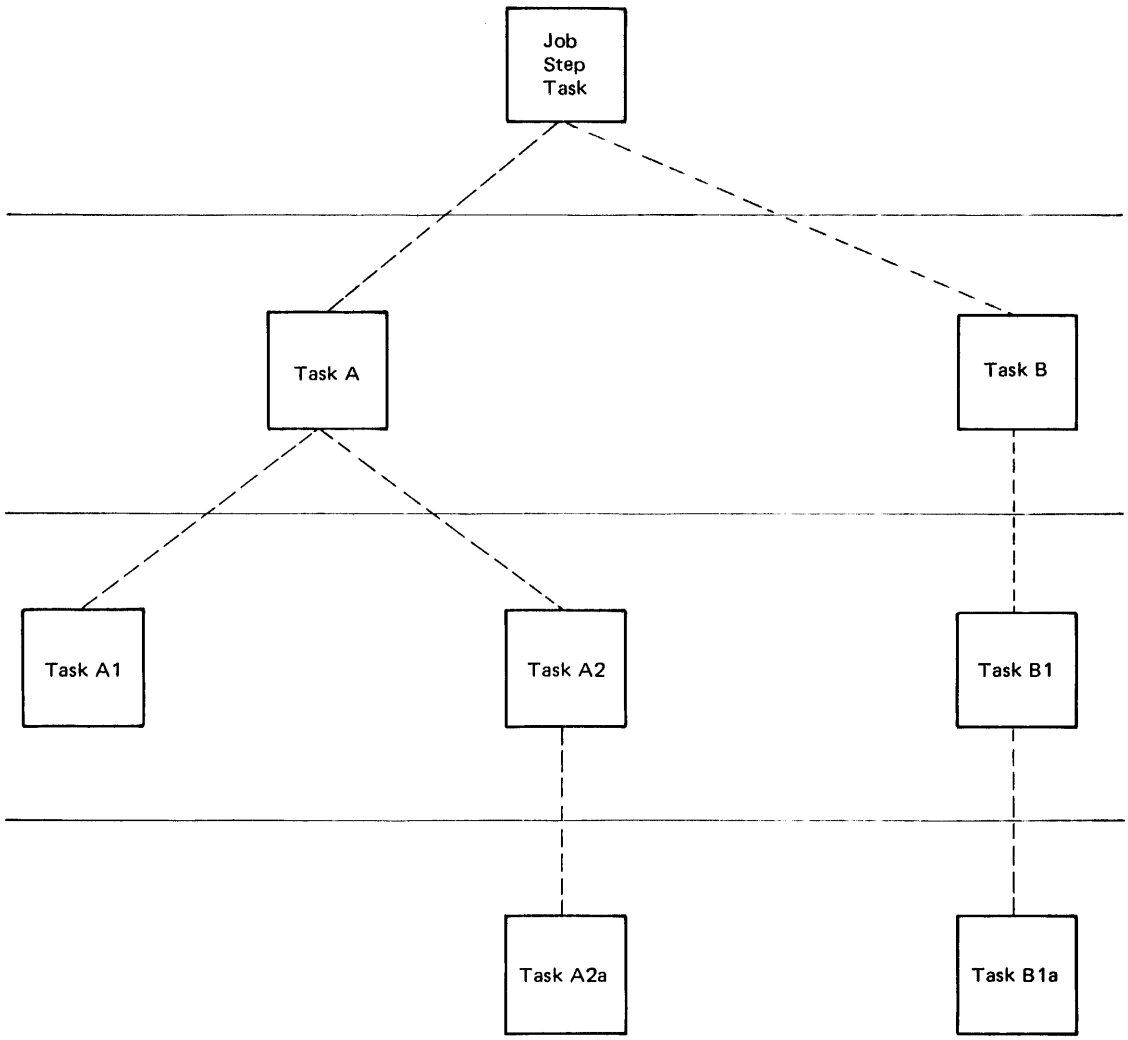


Figure 6. Levels of Tasks in a Job Step

All of the tasks in the job step compete independently for CPU 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. This is accomplished by issuing a DETACH macro instruction. 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 which has been removed from the system, which would cause the active task to be abnormally terminated.

> What if the originating task is not in wait state? See also p. 75.

Program Management

This chapter discusses facilities that aid you in designing your programs. Included are descriptions of load module structures, facilities for passing control between programs and the use of associated macro instructions.

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 7.

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 7. 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, any program should be carefully designed 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 VS operating system unless an ATTACH macro instruction is used to create a new task. The new task competes for CPU 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 *OS/VS - DOS/VS - VM/370 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 which establishes values, initializes switches, and acquires buffers for the other control sections in the program. The following procedures should be used 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 to the control section that makes 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.

Register 1 should be used to pass parameters. A parameter list should be established, and the address of the list placed 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 in the three low-order bytes of the word. The high-order bit of the last word should be set 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 are aware of your special conventions.

Since you have reloaded all the necessary registers, the save area that you used is now available, and can be reused by the called control section. You 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, and unnecessary, save area.

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 name.

An example of loading registers and passing control is shown in Figure 8. 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)       CSECT
                L       15,NEXTADDR     ENTRY NEXT
                LM      0,12,20(13)     ...
                BR      15-------->     NEXT SAVE (14,12)
                ...                     ...
NEXTADDR        DC      V(NEXT)
```

Figure 8. Passing Control in a Simple Structure

An example of passing a parameter list is shown in Figure 9. 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 9. 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. (Since one of these registers was the base register, restoring the registers earlier would have made the parameter list unaddressable.) A branch 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. The following procedures should be used when passing control with return.

Preparing to Pass Control

Registers 15 and 1 are used 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 the address of that save area should be passed 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 or a branch and link instruction. The other method uses the CALL macro instruction to provide a parameter list and establish the entry and return addresses. Using either method, the entry point must be identified by an ENTRY instruction in the called control section if the entry name is not the same as the control section name. Figures 10 and 11 illustrate the two methods of passing control; in each example, it is assumed that register 13 already contains the address of a new save area.

```
            ...
            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      B'10000000' Last parameter bit on
            DC      AL3(AREA)   Answer area address
NEXTADDR    DC      V(NEXT)     Address of entry point
GOOUT       BALR    14,15       Pass control; register 14 contains
                                return address
RETURNPT    ...
AREA        DC      12F'0'      Answer area from NEXT
```

Figure 10. Passing Control With Return

Use of an inline parameter list and an answer area is also illustrated in Figure 10. 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 10 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 first byte of the last address parameter (ANSWERAD) is coded with the high-order bit 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.

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

Figure 11. Passing Control With CALL

The CALL macro instruction in Figure 11 provides the same functions as the instructions in Figure 10. 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. Refer to the *Linkage Editor and Loader* publication, if you are interested in finding out more about this service.

The parameter list constructed from the CALL macro instruction in Figure 11, 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 control is returned from the control program after processing a system macro instruction, 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 4, so a branching table can be used easily, and a return code of 0 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 12 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 13, 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 12. 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 13. 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; control should always be returned to that address. You can either use a branch instruction such as BR 14, or you can use the RETURN macro instruction. An example of each method of returning control is discussed in the following paragraphs.

Figure 14 is 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 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

14 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'
```

Figure 14. Establishing a Return Code

The RETURN macro instruction is provided to save coding time. The expansion of the RETURN macro instruction provides instructions that restore a designated range of registers, load return code in register 15, and branch to the address in register 14. In addition, the RETURN macro instruction can be used to flag the save area used by the returning control section; this flag, a byte containing all ones, is placed in the high-order byte of word four of the save area 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. You will find that 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.

The contents of register 13 must be restored before the RETURN macro instruction is issued. The registers to be reloaded should be coded in the same order as they would have been designated had a load-multiple (LM) instruction been coded. 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 15 provides the same result as the coding shown in Figure 14. 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'
```

Figure 15. Using the RETURN Macro Instruction

Figure 16 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 16. 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 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. 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 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, 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 may 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 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 data management load modules; 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 requesting the load module. The parameters that define the order of the search are EP, EPLOC, DE, DCB, and TASKLIB. The EP, EPLOC, and DE parameters are used to specify 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 DCB parameter

is used to indicate the address of the data control block for the library containing the load module, and is optional. Omitting the DCB parameter or using the DCB parameter with an address of zero specifies the data control block for the link library or the job or step library. The TASKLIB parameter is used only for ATTACH.

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 17 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 antecedent 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 17. 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 that which 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 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.

This is not the case for task libraries. Extreme caution should be used when specifying module names in unique task libraries, because duplicate names may lead to the wrong module being given to the task requesting that the module be brought into virtual storage. Once a module has been loaded, the module name is known to all tasks in the region and a copy of that module is given to all tasks requesting that that module name be loaded, 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 18.

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

Figure 18. 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 used 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 order of the search when the DE parameter is used is shown in Figure 19 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 the link pack and job pack areas using the member name to determine if a usable copy of the load module exists in virtual storage. 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 for the region 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 for the region is searched for an available copy.
    The module is obtained from the step library; if there is no step library, the module is obtained from the job library.
DCB Parameter Indicates Private Library
    The job pack area for the region is searched for an available copy.
    The module is obtained from the specified private library.

Figure 19. 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 Figures 17, 18, and 19. 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 you. 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

The LOAD macro instruction is used to ensure that a copy of the specified load module is in virtual storage in your region or job pack area if it was not preloaded into the link pack area. When a LOAD macro instruction is issued, the control program searches for the load module as discussed previously and brings a copy of the load module into the region if required. When the control program returns control, register 0 contains 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. Normally, the LOAD macro instruction is used only for a reenterable or serially reusable load module, since the load module is retained even though it is not in use.

The control program also 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 or branch and link 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. 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.

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 placing the address of this parameter list in register 15. 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 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.

Figures 20 and 21 show the coding of a LINK macro instruction used to pass control to an entry point in a load module. In Figure 20, the load module is from the link, job, or step library; in Figure 21, 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 17.

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

Figure 20. 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 21. Use of the LINK Macro Instruction With a Private Library

Figures 22 and 23 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 22. Use of the BLDL Macro Instruction

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

Figure 23. The LINK Macro Instruction With a DE Parameter

The first parameter of the BLDL macro instruction is a zero, which indicates that the directory entry is on the link or job library. The second parameter is the address in virtual storage of the list description field for the directory entry. The first two bytes at LISTADDR indicate the length of each entry. If the entry is to be used in a LINK, LOAD, ATTACH, or XCTL macro instruction, the entry must be 60 bytes in length. 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 23 can now be written. Note that the DE parameter refers to the name field, not the list description field, of the directory entry.

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 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, or a CALL macro instruction can be used to pass control, using register 15. The return will be made directly to your program.

Note: 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. The best way to pass control is to use a CALL macro instruction or a branch or branch and link instruction.

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 or a branch and link instruction or a CALL macro instruction only.
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, and passing control using the address in register 14. 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, the register 15 to contain a return code. Control can be returned using a branch 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. 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 returned in register 0 is 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. A branch instruction is issued to pass control to the address in register 15.

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, with the address of the list in register 1. 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 24 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 24 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.

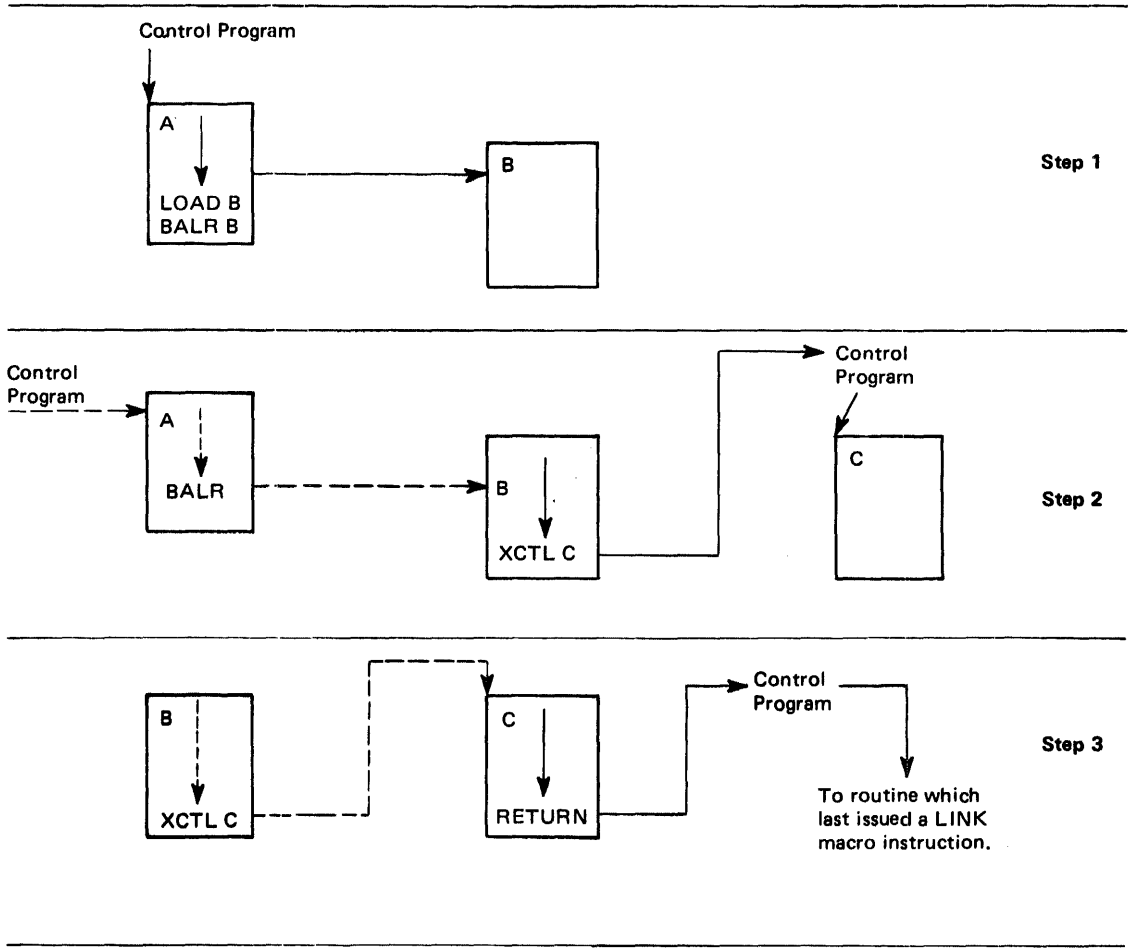


Figure 24. Misusing Control Program Facilities Causes Unpredictable Results

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 this restriction. The load module copy to which the entry point is to be added must be one of the following:

- A copy which 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. An IDENTIFY macro instruction can be issued by any program in the job step except by 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 a 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 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 08 if a duplicate is found. You are responsible for not duplicating a member name or an alias in any of the libraries.

Entry Point and Calling Sequence Identifiers as Debugging Aids

An entry point identifier is a character string of up to 70 characters which 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 which 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.

Task Synchronization

Some planning on your part is required to determine what portions of one task are dependent on the completions 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 25.

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 25. 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).

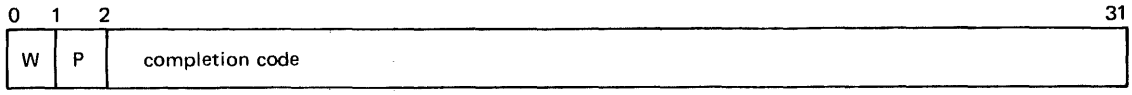


Figure 25. Event Control Block

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 wish to use a long wait when waiting for an ENQ or when using a timer interval.

Using a Serially Reusable Resource

When one or more users of a serially reusable resource modify the resource, simultaneous use must be prevented. Consider a data area in virtual storage that is being used by programs associated with several tasks of a job step. Some of the users are only reading records in the data area; since they are not changing the records, their use of the data area can be simultaneous. Other users of the data area, however, are reading, updating, and replacing records in the data area. Each of these users must acquire, update, and replace records one at a time, not simultaneously. In addition, none of the users that are only reading the records wish to use a record that another user is updating until after the record has been replaced. This illustrates why special care must be taken with serially reusable resources.

For all of the uses of the serially reusable resource made during the performance of a single task, you must prevent incorrect use of the resource yourself. You must make sure 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; since exit routines are given control asynchronously from the standpoint of your program logic, the exit routine could obtain a resource already in use by the main program. For the uses of the serially reusable resource by more than one task, the ENQ macro instruction is provided to ensure that the resource is used serially. The ENQ macro instruction cannot be used to *prevent* simultaneous use of the resource within a single task. It can only be used to *test* for simultaneous use within one task.

The ENQ macro instruction requests the control program to assign control of a resource to the active task or another task. The control program determines the status of the resource, and either grants the request by returning control to the active task, delays assignment of control by placing the active task in the wait condition, or passes back a return code indicating the status of the resource. When the status of the resource changes so that control can be given to a waiting task. the task is taken out of the wait condition and placed in the ready condition. The use of the ENQ macro instruction is discussed in the following paragraphs.

Naming the Resource

You represent the resource in the ENQ macro instruction by two names known as the qname and the rname, and by a scope indicator. These names may or may 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. It is up to all users of the resource to 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 resource. Because the actual resource is not identified by the control program, it is possible to use the resource without issuing an ENQ macro instruction requesting it. If this happens, the control program cannot provide any protection.

If the resource is used only in the performance of tasks in your job step, you should code the STEP parameter in the ENQ macro instructions that request the resource, indicating that the resource is used only in that job step. The control program adds the address space identifier to the scope so that duplicate qname and rname combinations can be used in different address spaces. If the resource is available to any address space in the system, the qname, rname, and scope combination must be agreed upon by all users. The SYSTEM parameter should be coded in each ENQ macro instruction requesting one of these resources.

When selecting a qname for the resource, do not use SYS as the first three characters; qnames used by the control program start with SYS and cannot be used.

Exclusive and Shared Requests

You can request exclusive or shared control of the resources for a task by coding either E or S in the ENQ macro instruction. If this use of the resource will result in modification of the resource, you *must* request exclusive control. If you are requesting use of a serially reusable load module and passing control yourself, you must request exclusive control, since that program modifies itself during execution. If you are updating a record in a data area, you must request exclusive control. If you are only reading a record, and you will not change the record, you can request shared control.

In order to protect any user of a serially reusable resource, all users must request exclusive or shared control on this basis: When a task is given control of a resource in response to a shared request, control will be given to other tasks simultaneously only in response to other requests for shared control, never in response to requests for exclusive control. A request for shared control will protect against modification of the resource by another task only if the above rules are followed.

Processing the Request

The control program constructs a list for each qname, rname, and scope combination it receives in an ENQ macro instruction, and enters a request in the list for the task which is active when the ENQ macro instruction is issued. The request is entered in an existing list when the control program receives a request specifying a qname, rname, and scope combination for which a list exists; if no list exists for that qname, rname, and scope combination, a new list is built. The requests are placed on the list in the order they are received by the control program; the priority of the task has no effect in this case. Control of the resource is allocated to a task according to two factors:

- The position on the list of the task's request.
- The exclusive control or shared control requirements of the request which caused the entry to be added to the list.

Figure 26 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 shared or exclusive control. The task represented by the first entry on the list is always given control of the resource, so the task represented by ENTRY1 (Figure 26, Step 1) is assigned the resource. The request which 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.

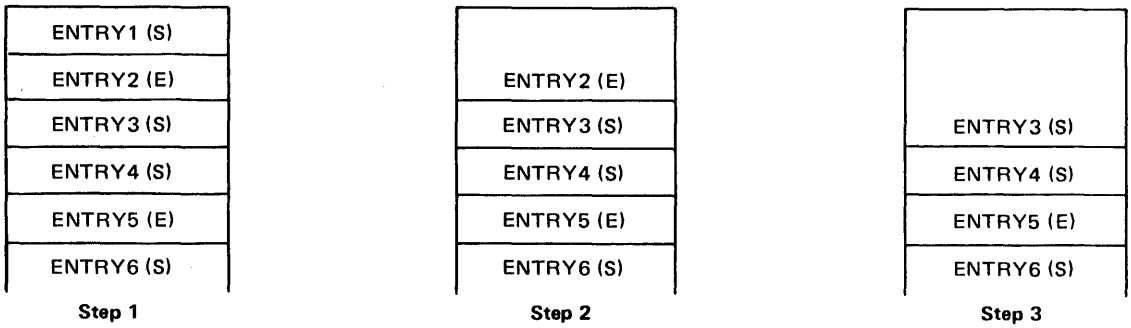


Figure 26. ENQ Macro Instruction Processing

Eventually, control of the resource is released for the task represented by ENTRY1, and the entry is removed from the list. As shown in Figure 26, Step 2, ENTRY2 is now first on the list, and the corresponding task is assigned control of the resource. Because the request which established ENTRY2 was for exclusive control, the tasks represented by all the other entries in the list are kept in the wait condition.

Figure 26, Step 3, shows the status of the list after control of the resource is released for the task represented by ENTRY2. Because ENTRY3 is now at the top of the list, the task represented by ENTRY3 is given control of the resource. ENTRY3 indicates that the resource can be shared, and, because ENTRY4 also indicates that the resource can be shared, ENTRY4 is also given control of the resource. In this case, the task represented by ENTRY5 will not be given control of the resource until control has been released for both the tasks represented by ENTRY3 and ENTRY4.

The following general rules are used by the control program:

- A task represented by the first entry in the list is always given 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 for an exclusive request are first in the list and all of the entries for a shared request are either first in the list or are preceded only by entries for other shared requests.

Using ENQ and DEQ

Proper use of the ENQ and DEQ macro instructions is required to avoid duplicate requests, to avoid tying up the resource, and to avoid interlocking the system. Guides to using them properly are given in the following paragraphs.

Duplicate Requests for a Resource

A duplicate request occurs when an ENQ macro instruction is issued to request a resource and a task has already been assigned control of that resource. If the second request results in a second entry on the list, the control program recognizes the contradiction and refuses to place the task in the ready condition (for the first request) and in the wait condition (for the second request) simultaneously. The second request results in a return code or abnormal termination of the task. You should design your program to ensure that a second request for a resource is never issued until control of the resource is released for the first use. Again, be especially careful when using an ENQ macro instruction in an exit routine.

Releasing the Resource

The DEQ macro instruction is used to release a serially reusable resource assigned to a task through the use of an ENQ macro instruction. The task must be in control of the resource. A resource cannot be released if the task does not have control. As you have seen, it is possible for many tasks to be placed in the wait condition while one task is assigned control of the resource. This may reduce the amount of work being done by the system. Issue a DEQ macro instruction as soon as possible to release the resource, so that other tasks can be performed. If a task returns control to the control program without releasing a resource, the resource is released automatically.

Conditional and Unconditional Requests

The normal use of the ENQ and DEQ macro instruction is to make unconditional requests, and these are the only requests that have been considered to this point. As you have seen, abnormal termination of the task occurs when two ENQ macro instructions are issued for the same resource in performance of the same task or subtask, without an intervening DEQ macro instruction. Abnormal termination also occurs if a DEQ macro instruction is issued in a task that has not been assigned control of the resource. Both of these abnormal termination conditions can be avoided either by careful program design or through the use of the RET parameter in the ENQ and DEQ macro instructions. The RET parameter (RET=TEST, RET=USE, RET=CHNG, and RET=HAVE for ENQ; RET=HAVE for DEQ) indicates a conditional request for or release of a resource.

RET=TEST is used to test the status of the list for the corresponding qname, rname, and scope combination. An entry is never made in the list when RET=TEST is coded. Instead, a return code is provided indicating the status of the list at the time the request was made. A

return code of 8 means the task is queued and has control of the resource. A return code of 20 means the task is queued but does not have control of the resource. A return code of 4 indicates the task would have been placed in the wait condition if the request had been unconditional. A return code of 0 indicates the task would have been given immediate control of the resource if the request had been unconditional. RET=TEST is most useful for determining if the task has already been assigned control of the resource. It is less useful for determining the status of the list and to take 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 been changed.

RET=USE indicates to the control program that the active task is to be assigned control of the resource only if the resource is immediately available. A return code of 0 indicates that the request was put on the list and the task was assigned control of the resource. A return code of 4 indicates that the task would have been placed in the wait condition if the request had been unconditional. A return code of 8 means the task is queued and has control of the resource. A return code of 20 means the task is queued but does not have control of the resource. The request is not put on the list if any return code other than 0 is given. RET=USE can be best used when there is other processing that can be done without using the resource. You do not want to wait for the resource if there is other work that you can do.

RET=CHNG indicates to the control program that the caller wishes to have exclusive control of a resource which he has already requested. A return code of 0 indicates that the resource was available and was assigned to the exclusive control of the caller. Either the caller had already requested exclusive control, or the requested change from shared to exclusive control was honored. A return code of 4 indicates that the requested change in attribute cannot be honored, because the caller is currently sharing the resource with another user. A return code of 8 indicates that the user was not queued to receive control of the resource when he requested the attribute change. Although this is an error condition, control is returned to the user. A return code of 20 means the task is queued but does not have control of the resource.

RET=HAVE is used in both the ENQ and DEQ macro instructions. An ENQ macro instruction is treated as a normal request for control unless a request from the same task already exists. A return code of 8 means the task is queued and has control of the resource. A return code of 20 means the task is queued but does not have control of the resource. A return code of 0 indicates that the task was assigned control of the resource. A DEQ macro instruction is processed as a normal request to release a resource unless the task does not have control of the resource. A return code of 0 indicates that the resource has been released. A return code of 8 indicates that the task did not have an entry for the resource. RET=HAVE can be used to good advantage in an exit routine to avoid abnormal termination.

Avoiding Interlock

An interlock condition arises when two tasks are waiting for each others' completion, yet neither task can gain access to the resource necessary for its completion. An example of an interlock is shown in Figure 27. Task A has exclusive access to resource M, and higher-priority task B has exclusive access to resource N. Task B is placed in a wait condition when it requests exclusive access to resource M because M is accessible only by task A. The interlock becomes complete when task A requests exclusive access to resource N, because N is accessible only by Task B. 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 27. 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 interlocks be avoided. The following procedures indicate some ways of preventing interlocks.

- Do not request resources that are not immediately required. If you can use the serially reusable resources one at a time, you should 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 27 had been for shared resources, 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 would suffice.
- The ENQ macro instruction can be written to request control of more than one resource at a time. The requesting program is placed in a wait state 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 and are unavailable to any other programs that may request them. For example, instead of coding the two ENQ macro instructions shown in Figure 28, the one ENQ Macro instruction shown in Figure 29 could be coded. If all requests were made in this manner, the interlock shown in Figure 27 would be avoided. All of the requests from one task would be processed before any of the requests from the second task. The DEQ macro instruction should release a resource as soon as it is no longer needed.

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

Figure 28. Two Requests For Two Resources

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

Figure 29. One Request For Two Resources

- If the use of one resource *always* depends on the use of a second resource, then the pair of resources can be defined as one resource in the ENQ and DEQ macro instructions. This procedure can be used for any number of resources that are always used in combination. There would be no protection of the resources if they are also requested independently, however. The request would always have to 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 the order in which control of the resources is requested should be the same for each user. For instance, if resources A, B, and C are required in the performance of 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.

The above is not an exhaustive list of the procedures to be used to avoid an interlock. You could also make repeated requests for control specifying the RET=USE parameter, which would prevent the task from being placed in the wait condition; if no interlock was developing, of course, this would be a waste of execution time. The solution to the interlock problem in all cases requires the cooperation of all the users of the resources.

The supervisor offers many services to assist in the detection and processing of abnormal conditions during the execution of the system. Certain types of abnormal conditions are detected by the hardware and cause program interruptions to occur (for example, if an attempt is made to execute an instruction with an invalid operation code). Other abnormal conditions are detected by the software (for example, an attempt to open a data set which is not defined to the system causes an ABEND to be issued by the Open routine).

Although the supervisor provides facilities for writing exit routines to handle specific types of interruptions and abnormal conditions, the supervisor 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.

Program Interruption Processing

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.* For certain exceptions (fixed-point and decimal overflow, exponent underflow and significance), interruptions can be disabled by setting the corresponding bits in the program status word to zero.

When a task becomes active for the first time, all program interruptions that can be disabled are disabled, and a standard control program exit routine, included when the system was generated, is provided. This control program exit routine is given control when certain program interruptions occur; it issues an ABEND macro instruction specifying task abnormal termination and requesting a dump. By issuing the SPIE macro instruction, you can specify your own exit routine to be given control for one or more types of program exceptions. The macro instruction specifies the address of the exit routine to be given control when specified program exceptions occur. If the SPIE 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 SPIE macro instruction can be issued by any problem program being executed in performance of the task. When the task is active, your exit routine receives control for all interruptions resulting from exceptions specified in the SPIE macro instruction unless the current routine for the task is operating in supervisor mode. For other program interruptions, control is given to the control program exit routine. Each succeeding SPIE macro instruction completely overrides specifications in the previous macro instruction.

Program Interruption Control Area

The expansion of each standard or list form SPIE macro instruction contains a control program parameter list called the program interruption control area (PICA). The PICA and another control program area called the program interruption element (PIE) contain the information that enables the control program to intercept user-specified program interruptions. Together, the PIE and the PICA associated with it are called the "SPIE environment." (The PIE is described later in this section.) The PICA, as shown in Figure 30, contains the new program mask for the interruption types that can be disabled, 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.

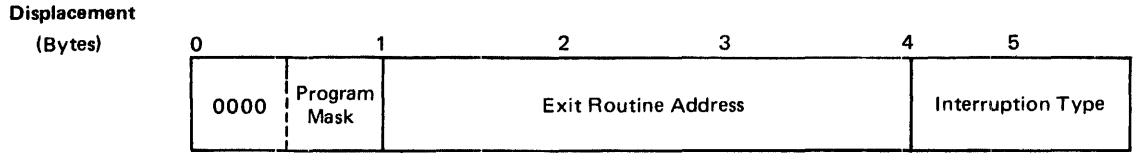


Figure 30. Program Interruption Control Area

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

The effect of the last SPIE macro instruction is canceled 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 zeros, thus indicating that no user exit routine is to process interruptions. Any previous SPIE environment may be reestablished, regardless of the number or type of subsequent SPIE macro instructions issued, by using the execute form of the SPIE macro instruction specifying the appropriate value that had been returned in register 1 by the control program. If a PICA address is specified (as opposed to zeros), the PICA must still be valid (not overlaid). The SPIE environment will be eliminated by specifying zeros as the PICA address.

Figure 31 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 31. Using the SPIE Macro Instruction

Program Interruption Element

At the first execution of 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 the job step. Because the PIE is freed when the SPIE environment is eliminated (by specifying a PICA address of zero in the execute form of a SPIE macro instruction), a PIE will also be created whenever a SPIE macro instruction is issued and no PIE exists. The format of the PIE is shown in Figure 32.

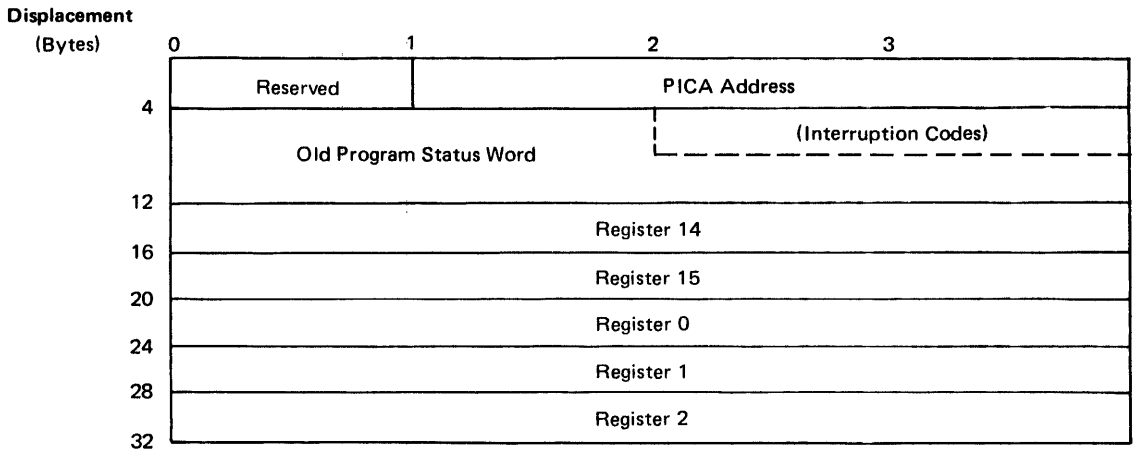


Figure 32. Program Interruption Element

The PICA address in the program interruption element is the address of the program interruption control area used in the last execution of a SPIE macro instruction for the task. When control is passed to the routine indicated in the PICA, the old program status word contains the interruption code in bits 16-31; these bits can be tested to determine the cause of the program interruption. The contents of registers 14, 15, 0, 1, and 2 at the time of the interruption are stored by the control program as indicated.

Register Contents Upon Entry to User's Exit Routine

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

Register 0: Internal control program information.

Register 1: Address of the program interruption element 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 must not use this save 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. 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 is given control.

To determine which type of interruption occurred, the exit routine can test bits 28 through 31 of the old program status word (OPSW) in the program interruption element. The routine can then take corrective action or can simply ignore the exceptional condition.

The exit routine can alter the contents of the registers when control is returned to the interrupted program. For registers 3 through 13, the routine alters the contents of the actual registers. For registers 14 through 2, the routine alters the contents of the register save area in the program interruption element, because the control program reloads these registers from this area when it returns control to the interrupted program. The exit routine can also alter the last four bytes of the OPSW in the program interruption element. By changing the OPSW, the routine can select any return point in the interrupted program.

Handling Abnormal Conditions

It is not possible to provide procedures for all possible conditions which can occur during the execution of a program. You should, of course, be sure that you can process all valid data, and that your program satisfies all the requirements of the problem. The more general you make the program, the greater the number of additional routines you will require to handle special cases. But you will not be able to provide routines to detect and correct all of the special or abnormal conditions that can occur.

The control program does a great deal of checking for abnormal conditions. A standard program interruption routine is provided to detect and process 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 no requests with conflicting requirements have been made. For the abnormal conditions that can possibly be corrected, control is returned 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 existing data, the control program abnormal termination routine is given control.

There will be abnormal conditions unique to your program, of course, that the control program cannot detect. Figure 33 is an example of one of these. The routine shown in Figure 33 checks a control field in an input parameter list to determine which function the program is to perform. Only characters between 1 and 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. The routine should indicate its inability to continue processing by returning control to the calling program with an error return code. The calling program should then try to interpret the return code and to recover from the error. If it cannot do so, the calling program should detach its incomplete subtasks, execute its usual termination procedures, and return control to its calling program, again with an error return code. This procedure may result in termination of all the tasks of a job step; if it does, the COND parameters of the JOB and EXEC statements may be used to determine whether subsequent job steps should be executed.

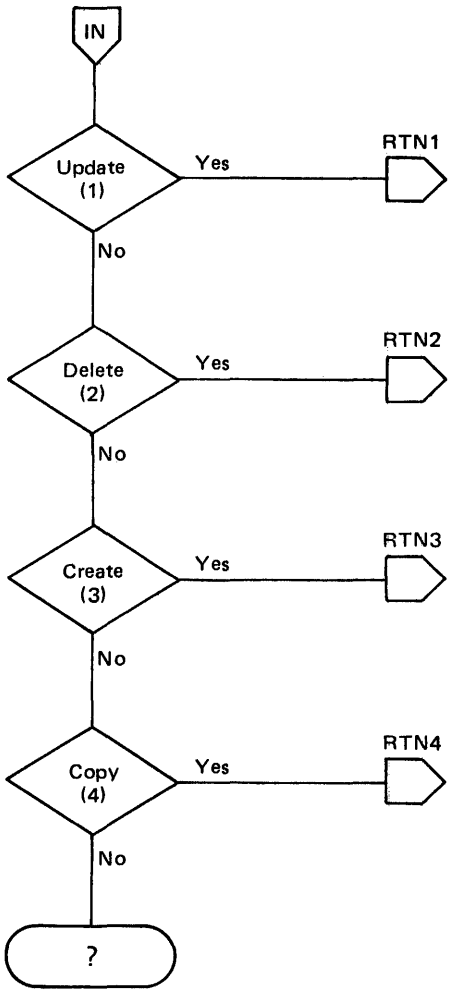


Figure 33. Detecting an Abnormal Condition

An alternative to this procedure is to pass control to the control program abnormal termination routine by issuing an ABEND macro instruction. In this case, if an error exit routine was established via the ESTAE macro instruction or the ATTACH macro instruction with the ESTAI or STAI option, the error exit routine gets control. (This error exit routine also receives control if the system issues an ABEND macro instruction.) The exit can then determine its actions with regard to the abnormal condition. This approach permits the implementation of mainline routines which contain less error handling code (for example, there is no need to check return codes after invocation of a subroutine if the subroutine issues an ABEND). The error handling functions can be packaged in the ESTAE/ESTAI exits which execute only when an error occurs.

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. An ABEND macro instruction (without a STEP parameter) that is issued in performance of any task other than the job step task usually causes only that task and the subtasks of that task to be abnormally terminated. However, if the abnormal termination cannot be fulfilled as requested, it may be necessary for the control program to abnormally terminate the job step task. The

abnormal termination routine works in the same manner whether it is given control from the control program or a problem program.

If the job step *is not* to be terminated, the following actions are taken:

- The resources owned by the terminating task and all of its subtasks are released, starting with the lowest level task.
- The (system or user) completion code specified in the ABEND macro instruction is placed in the task control block of the active task (the task for which the ABEND macro instruction was issued).
- If the ECB parameter was written in the ATTACH macro instruction issued to create the active task, the ECB is posted with the completion code specified in the ABEND macro instruction.
- If the ETXR parameter was written in the ATTACH macro instruction issued to create the active task, the end-of-task exit routine is scheduled to be given control when the originating task becomes active.
- If neither the ECB nor ETXR parameter was written when the ATTACH macro instruction was issued, a DETACH macro instruction is issued by the control program for the active task.

If the job step *is* to be terminated, the following actions are taken:

- The resources owned by each task are released, starting with the lowest level task, for all tasks in the job step. No end-of-task exit routine is given control.
- The (system or user) completion code specified in the ABEND macro instruction is written on the system output device.
- Unless you specify otherwise in your job control statements, the remaining job steps in the job are skipped. However, the statements defining these steps are checked for proper syntax.

It is possible to restart a job step that has been abnormally terminated. Restart can occur either at the beginning of the job step or at an internal checkpoint. A detailed discussion of checkpoint and restart appears in *Checkpoint/Restart.*

Intercepting Abnormal Termination of Tasks

Abnormal termination of a task can be intercepted through the use of the ESTAE macro instruction. When a task that has previously issued an ESTAE macro instruction is scheduled for abnormal termination, control is passed to the user at his ESTAE exit routine address. Within the ESTAE exit routine, the user can perform pre-termination functions and diagnose the error. He can also determine whether abnormal termination should continue for the task, or whether normal processing can resume at some retry point.

When the abnormal termination is scheduled, the ESTAE exit routine must be resident. It may either be part of the program issuing ESTAE or be brought into virtual storage via the LOAD macro instruction.

A single user program can issue more than one ESTAE macro instruction with the CT (create) parameter. All ESTAE requests issued by programs running under the same task are queued so that the exit established by the most recent ESTAE request will be the first to get control. If this exit fails or requests that the abnormal termination continue, the exit established by the previous ESTAE request will get control.

If the user wishes to use the same exit routine for several tasks at the same time, it must be reenterable. For convenience, it is recommended that all ESTAE exit routines be reenterable.

The user can cancel or overlay the current ESTAE request; that is, the one most recently made. If no ESTAE requests are active for the task when a cancel or overlay is issued, or if the user attempts to cancel or overlay an ESTAE request not associated with his request block level of control, he will be informed that his request is invalid by a return code. An ESTAE request can be canceled by issuing the ESTAE macro instruction with the ESTAE exit routine address specified as zero. Overlaying is done by issuing an ESTAE macro instruction specifying OV. Every program should cancel all ESTAE exits it has created before returning control to its caller.

Interface to an ESTAE exit

Before the initial ESTAE exit routine receives control, the I/O and asynchronous processing requests specified in the ESTAE macro instruction are fulfilled. The I/O processing requests will be performed only for the first exit selected; subsequent exits, if entered, will receive an indication of the I/O status, but no additional I/O processing will be performed. The asynchronous exit processing requests, however, will be fulfilled for each exit.

Before each ESTAE exit receives control, the control program attempts to obtain and initialize a work area which will control information about the error. This work area is called the System Diagnostic Work Area (SDWA). (The SDWA mapping macro - IHASDWA - must be included in the routine.) The first word of the SDWA contains the address of the parameter list established via the ESTAE macro instruction. If the SDWA is obtained, the contents of the registers on entry to the ESTAE exit routine are:

register 0	A code indicating the type of I/O processing performed: 0 — Active I/O has been quiesced and is restorable. 4 — Active I/O has been halted and is not restorable. 8 — No I/O was active at time of ABEND. 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.

The exit routine is enabled and has the same protection key as the routine which established the exit as long as that routine was under a problem program protection key (keys 8-15). An ESTAE exit created by a program running under any other control program protection key (keys 0-7) receives control in key 0. Entry is made to the ESTAE exit via the SYNCH macro instruction.

When the ESTAE exit has completed its analysis of the error, it should use the SETRP macro instruction to inform the control program of the actions it desires. The SETRP macro instruction will initialize the SDWA with the desired options.

Return from the ESTAE exit can optionally be effected via the SETRP REGS parameter or by a BR 14 instruction.

If storage was not available for the SDWA, the register contents upon entry to the ESTAE exit routine are as follows:

register 0	12 (decimal).
register 1	ABEND completion code.
register 2	Address of the parameter list specified on the ESTAE macro instruction, or 0.
registers 3-13	Unpredictable.
register 14	Return address.
register 15	Entry point address.

The exit routine is enabled and has the same protection key as the routine which established the exit as long as that routine was under a problem program protection key (keys 8-15). An ESTAE exit created by a program running under any other control program protection key (keys 0-7) receives control in key 0. Entry is made to the ESTAE exit via the SYNCH macro instruction.

If the control program could not provide a work area, a register save area will not be provided either. In this case, register 14 must be saved and used as the return register to the control program.

If a work area (SDWA) was not provided, the user must place a return code in register 15 before returning control to the control program from the ESTAE exit routine. The return code indicates whether ABEND processing is to be continued for the task or whether termination can be circumvented and a retry address given control. The return codes placed in register 15 may be:

0 — Termination should be continued. (Any ESTAE exits that were established prior to this exit will receive control.)
4 — A retry address is provided. (This address must be placed in register 0.)

The ESTAE exit routine returns control via BR 14.

Intercepting Abnormal Termination of Subtasks

To provide an exit in your program to intercept abnormal termination of a subtask, use the ESTAI parameter of the ATTACH macro instruction you issue to create the subtask. The ESTAI request issued for any subtask will be extended to all subtasks. For example, suppose task A attaches task B and uses the ESTAI parameter of ATTACH. When task B attaches task C, the ESTAI request issued by A will be active for C as well as B.

Since more than one subtask abnormally terminate at the same time, the ESTAI exit routine may be used by more than one task concurrently. Therefore, the exit routine must be reenterable.

Interface to an ESTAI exit

ESTAI exits are entered after all ESTAE exits that exist for a given task have received control and have either failed or requested that the termination continue.

The interface to ESTAI exits is the same as that for ESTAE exits. However, one additional option is available for ESTAI. In relinquishing control to the system, return code 16 may be specified either on the SETRP macro instruction if an SDWA exists or in register 15 if an SDWA is not available. The return code means that the termination should be continued and no further ESTAI exits should receive control for that task.

ESTAE/ESTAI Retry Routines

If a given ESTAE/ESTAI exit routine requests that the termination be continued, the control program will give control to the next oldest ESTAE/ESTAI exit which exists for the task. However, if a given ESTAE/ESTAI exit routine requests that a retry address be given control, a dump will be taken if requested (unless suppressed by the exit), and no further ESTAE/ESTAI exits will be processed. Instead, the address specified as the retry address will be given control.

The ESTAE/ESTAI retry routine, like the ESTAE/ESTAI exit routine must be in virtual storage when the exit routine determines that retry is to be attempted. If not already resident within your load module, it may be brought into storage via the LOAD macro instruction.

An ESTAE retry routine will execute under the request block that issued the ESTAE macro instruction; all newer request blocks will be purged before the retry routine is passed control.

An ESTAI retry routine will execute under the request block for the latest ESTAE or ESTAI exit routine. (A request block will exist for a previous ESTAE or ESTAI exit if one had abnormally terminated during execution.) If no previous ESTAE or ESTAI exit has failed, the RB queue is purged until only program request blocks PRBs remain. Then, the retry routine will get control under the newest PRB left on the queue.

When control is passed to a retry address, the ESTAE macro instruction does not have to be reissued to continue to use the same ESTAE exit. However, ESTAE may be issued to add or change exits.

If an SDWA was passed to the exit and FRESDWA=YES was not specified on the SETRP macro instruction, the retry routine should issue the FREEMAIN macro instruction to free the storage occupied by the SDWA when it is no longer needed. The subpool number and the length which should be used on the FREEMAIN macro instruction are contained within the SDWA.

Interface to a Retry Routine

There are two different interfaces to the retry routine:

- If an SDWA was obtained, you can set in the SDWA the register contents you wish to have and request that they be passed to the retry routine by coding RETREGS=YES in the SETRP macro instruction. This alternative is most often used in mainline processing.
- If no SDWA was obtained or if RETREGS=NO was specified on the SETRP macro instruction, only parameter registers are passed to the retry routine. This alternative is most often used if a special retry routine is to get control.

The register contents are as follows:

If an SDWA was not obtained:

register 0	12.
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 retry routine.

If an SDWA was obtained and the exit did not request register update (RETREGS=NO) or release of the SDWA (FRESDWA=NO):

register 0	0.
register 1	Address of SDWA.
registers 2-13	Unpredictable.
register 14	Address of an SVC 3 instruction.
register 15	Entry point address of retry routine.

If an SDWA was obtained and the exit did not request register update but did request release of the SDWA:

register 0	20.
register 1	Address of the user parameter list established via the ESTAE macro instruction.
register 2	Address of the PIRL if I/O was quiesed and is restorable. Otherwise, 0.
registers 3-13	Unpredictable.
register 14	Address of an SVC 3 instruction.
register 15	Entry point address of retry routine.

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

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

Dumping Services

There are two types of storage dumps that can be requested by a problem program of the operating system:

- A 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 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.

This dump will be provided only if a SYSABEND or SYSUDUMP DD statement is included in the job step. If the DD statement is provided and dump options are requested, the dump will contain all requested areas in addition to the default dump options that were installation-selected for the SYSABEND or SYSUDUMP DD statement. If the DD statement is provided and no dump options are requested, only the installation-selected dump options will be provided.

There is one execption to the processing described above. If the operator issues the CHNGDUMP command, the command will override all options specified by the installation and all options specified during the abnormal condition processing.

If a dump is requested and the ESTAE/ESTAI exit also requests retry, the dump will be taken by the control program prior to passing control to the retry address.

The data set containing the dump can reside on any device which is supported by the basic sequential access method (BSAM). The dump is placed in the data set described by the DD statement you provide. If a printer is selected the dump is printed immediately. However, if a direct access or tape device is designated, a separate job must be scheduled to obtain a listing of the dump, and to release the space on the device.

SNAP Dumps

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

Like the ABEND dump, the data set containing the dump can reside on any device which is supported by BSAM. The dump can reside on any device which is supported by BSAM. The dump is placed in the data set described by the DD statement you provide. If a printer is selected, the dump is printed immediately. However, if a direct access or tape device is designated, a separate job must be scheduled 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 any SNAP macro instructions are issued. The data control block *must* contain the following parameters: DSORG=PS, RECFM=VBA, MACRF=W, BLKSIZE=882, and LRECL=125. (The data control block is discussed in the *Data Management Services* manual.) If your program is to be processed by the loader, you should also issue a CLOSE macro instruction for the SNAP data control block.

You obtain the use of 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* 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 four methods of explicitly requesting virtual storage using a GETMAIN macro instruction. Each of the methods, which are designated by coding an associated character in the parameter field of the GETMAIN macro instruction, has certain advantages, depending on the requirements of your program. The last three methods do not produce reenterable coding

unless coded in the list and execute forms, as indicated in "Implicit Requests." The methods are as follows:

Register Type: Specifies a request for a single area of virtual storage of a specified length. The address of the area is returned in register 1. This type of request produces reenterable coding, because parameters are passed to the control program in registers, not in a parameter list.

Element Type: Specifies a request for a single area of virtual storage of a specified length. The control program places the address of the allocated area in a fullword that you supply.

Variable Type: Specifies a request for a single area of virtual storage with a length between two values you specify. The control program 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. The control program places the address of the area and the length allocated in two consecutive fullwords that you supply.

List Type: Specifies a request for one or more areas of virtual storage 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 34. 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
                GETMAIN VU,LA=SIZES,A=ANSWADD   try to get 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 34. Using the GETMAIN Macro Instruction

Subpool Handling

A conditional request for a single element of storage with a length of 16,000 bytes is requested in Figure 34. 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.

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 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.

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 35 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 not sufficient unallocated area remaining in Block B (blocks are checked in the order first in, first out), but there is enough area in Block A, 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.

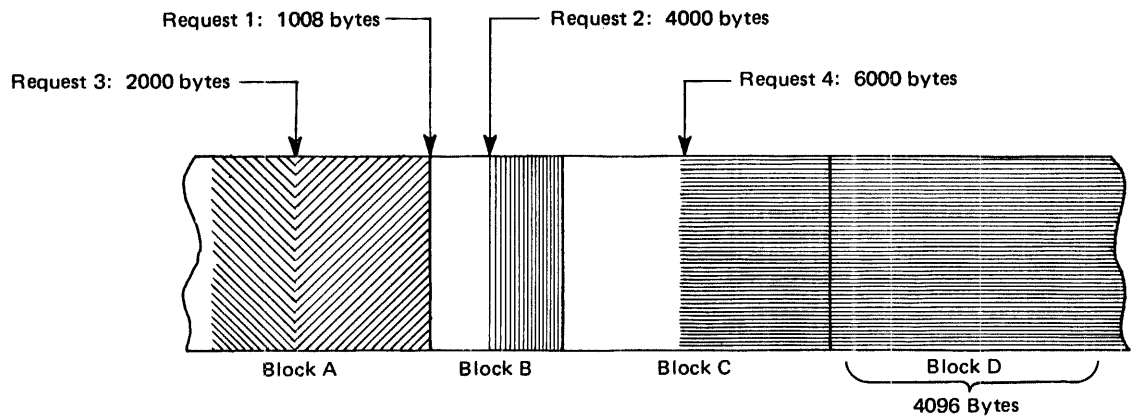


Figure 35. Virtual Storage Control

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.

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: A new subpool is created whenever SHSPV or SHSPL is coded on an ATTACH macro instructions or a GETMAIN macro instruction is issued, and the subpool(s) specified does not exist for the active task. A new subpool zero is created for the subtask if SZERO=NO is specified on ATTACH. If one of the ATTACH macro instruction parameters 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 ownership; if the active task is sharing a subpool and an attempt is made to pass ownership 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 it does not modify itself during execution. 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 which pass parameters in registers 1 and 0, and macro instructions which 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 POINT macro instruction requires that the DCB address and block address be coded as follows:

```
POINT dcb address,block address
```

One method of coding this in a reenterable program would be to require that both of these addresses refer to a portion of storage allocated to the active task through the use of a GETMAIN macro instruction. The addresses would change for each use of the load module. Therefore, you would load two of the general registers 2-12 with the addresses, and designate the appropriate registers when you code the macro instruction. If register 4 contains the DCB address and register 6 contains the block address, the POINT macro instruction is written as follows:

```
POINT (4),(6)
```

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 36 shows the user 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 255 bytes only. Some macro instructions do produce lists greater than 255 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.

```
            ...
            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 255
* 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'
```

Figure 36. Using the List and the Execute Forms of the DEQ Macro Instruction in a Reenterable Program

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.

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

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 can be released by issuing a FREEMAIN macro instruction 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's 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).

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 theactual 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 a pool of available frames, thereby associating virtuual 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 the 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 as 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 VS2 include the following:

- PGRLSE — Release virtual storage contents.
- PGLOAD — Load virtual storage areas into real storage.
- PGOUT — Page out virtual storage areas from real storage.

The PGRLSE 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 PGRLSE 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 PGLOAD and PGOUT 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. The PGRLSE macro makes available all real and external page storage wholly associated with the area of virtual address space specified. As shown in Figure 37 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, PGRLSE may improve operating efficiency.

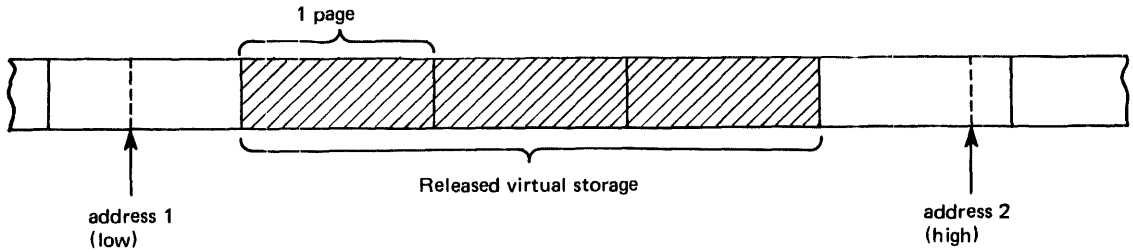


Figure 37. Releasing Virtual Storage

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 PGRLSE for any resulting free page.

Loading/Paging Out Virtual Storage Areas

The PGLOAD macro instruction essentially provides 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 PGLOAD, 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 a new real frames assigned without page-in operations; if you specify RELEASE=N, the contents are to remain intact and later used.

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

The PGOUT 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: 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; parameter list entry; 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.)	KEEPREAL 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.

Timing Services

Interval timing is a standard feature of VS. 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 supplied by you if the MIC or STCK parameter was specified.

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 *IBM System/370 Principles of Operation.)*

Interval Timing

A time interval, up to a maximum of 24 hours, can be established for any task in the job step through the use of the STIMER macro instruction, and the time remaining in the interval can be tested and canceled through the use of the TTIMER macro instruction. Each task in the job step can have an active time interval.

When you request a time interval, you also specify the manner in which the interval is to be decreased, through the use of the TASK, REAL, or WAIT parameter of the STIMER macro instruction. REAL and WAIT both indicate that the interval is to be decreased continuously, whether the associated task is active or not. TASK indicates that the interval is to be decreased only when the associated task is active. If REAL or TASK is coded, the task continues to compete with the other ready tasks for control; if WAIT is coded, the task is placed in the wait condition until the interval expires, at which time the task is placed in the ready condition.

When TASK or REAL is designated, the address of a timer completion exit routine can be specified. This is the first routine to be given control when the associated task is made active after the completion of the time interval. (If the address of the 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 required, and must save and restore registers and return control to the address in register 14. After control is returned to the control program, control is passed to the next instruction in the main program.

Figure 38 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 38. Interval Timing

The loop continues as long as the value in register 12 is less than or equal to the value in register 7. 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.

Extended-Precision Floating-Point Simulation

The System/370 Extended-Precision Floating-Point Simulator provides full extended-precision arithmetic for all VS users. A divide macro instruction (DXR) is provided for the models that have the extended-precision floating arithmetic facility and all seven instructions are provided for the models that do not. The instructions provided are:

Name	Mnemonic
ADD NORMALIZED (extended)	AXR
LOAD ROUNDED (extended to long)	LRDR
LOAD ROUNDED (long to short)	LRER
MULTIPLY (extended)	MXR
MULTIPLY (long to extended)	MXDR
MULTIPLY (long to extended)	MXD
SUBTRACT NORMALIZED (extended)	SXR

For more details on the instructions, see *System/370 Principles of Operation.*

Thus, you can use extended-precision floating-point instructions whether or not your particular machine model has the extended-precision floating-point facility. To do so, write a program-interruption-handling exit routine. The exit routine is required:

- If your machine model already has the extended-precision floating-point facility, and you also wish to use the extended-precision floating-point divide (DXR) macro instruction.
- If your machine model does not have the extended-precision floating-point instructions, but you wish to use these instructions and the extended-precision floating-point divide instruction.

To determine if the extended-precision floating-point feature is installed in your CPU, call the module IEAXPSIM, which returns a pointer to the appropriate simulator.

The format of the extended-precision floating-point divide (DXR) instruction is described in the macro instructions section, and the formats of the other extended-precision floating-point instructions are described in *System/370 Principles of Operation.*

Extended-Precision Division

To perform extended-precision division, use the DXR macro instruction:

```
DXR reg1,reg2
```

where reg1 contains the dividend, reg2 the divisor.

The first parameter (the dividend) is divided by the second parameter (the divisor) and is replaced by the normalized quotient. No remainder is preserved. For a discussion of normalization, refer to the section "Floating-Point Arithmetic" in *System/370 Principles of Operation.*

Division Process

The quotient fraction has 28 hexadecimal digits and is developed such that it is the largest number for which the absolute value of the product of the quotient and the divisor fractions is either equal to or less than the absolute value of the adjusted (normalized) dividend fraction. All digits of the dividend and divisor fractions are involved in the operation; the dividend fraction is extended with low-order zeros.

The sign of the quotient is determined by the rules of algebra; however, if the quotient is made a true 0, its sign is made plus.

Unless the quotient is made a true 0, the characteristic, sign, and high-order 14 hexadecimal digits of the normalized quotient fraction replace the high-order part of the first parameter. The low-order 14 hexadecimal digits of the quotient fraction replace the low-order fraction of the first parameter. The low-order sign is made equal to the high-order sign, and the low-order characteristic is made 14 less than the high-order characteristic. However, when the subtraction of 14 causes the low-order characteristic to become less than 0, it is made 128 greater than its correct value. Extended-precision arithmetic is further discussed in *System/370 Principles of Operation.*

Arithmetic Exceptions

The following exceptions can occur when using the DXR macro instruction.

- Exponent overflow.
- Exponent underflow.
- Floating-point divide.

Exponent overflow is recognized when the characteristic of the normalized quotient exceeds 127 and the fraction of the quotient is not 0. The operation is completed by making the high-order characteristic 128 less than the current value. If the low-order characteristic also exceeds 127, it is decreased by 128. The quotient fraction and sign remain unchanged. A program interruption for exponent overflow then occurs.

Exponent underflow is recognized when the characteristic of the normalized quotient is less than 0 and neither parameter fraction is 0. If the exponent underflow mask bit is set, the operation is completed by making the characteristics of both parts 128 greater than their correct values. The quotient fraction and sign remain unchanged. A program interruption for exponent underflow then occurs. If the exponent underflow mask is 0, a program interruption does not occur; instead, the operation is completed by making both the high-order and low-order parts of the quotient a true 0.

Exponent underflow is not recognized when the low-order characteristic is less than 0 and the high-order characteristic is greater than or equal to 0. Similarly, exponent underflow is not recognized when one or both of the parameters underflow during prenormalization, but the quotient can be expressed without encountering underflow.

The floating-point divide exception is recognized when the divisor fraction is 0. The operation is suppressed, and a program interruption for floating-point divide occurs.

When the dividend fraction is 0, the quotient is made a true 0, and a possible exponent overflow or underflow is not recognized. A division of 0 by 0, however, causes the operation to be suppressed and an interruption for floating-point divide to occur.

The condition code remains unchanged for all arithmetic exceptions. Figure 39 describes the program interruptions that can occur.

Interruption Type	Description	Action Taken
Operation	The instruction is not installed.	The operation is suppressed.
Specification	Registers other than 0 or 4 are specified, or positions 16-23 do not contain 0's.	The operation is suppressed.
Exponent Overflow	The characteristic of the normalized quotient exceeds 127, and neither operand fraction is 0.	The operation is completed.
Exponent Underflow	The characteristic of the normalized quotient is less than 0, neither operand fraction is 0, and the exponent underflow mask bit is set.	The operation is completed.
Floating-Point Divide	The divisor fraction is 0.	The operation is suppressed.

Figure 39. Summary of Program Interruptions

Calling the Simulator

To use the extended-precision floating-point instructions that your machine model does not have, call the extended-precision floating-point simulator from a program-interruption-handling exit routine. The simulator is a program that is automatically included in your operating system at system generation time. Writing an exit routine to handle program interruptions is discussed under "Program Interruption Processing."

To use the extended-precision floating-point simulator, specify in the SPIE macro instruction that your exit routine is to receive control if an operation exception occurs. In addition, the exit routine must perform the following tasks, in this order:

- Check that the exception is for floating-point divide.
- Prepare a parameter list to pass to IEAXPSIM.
- Pass control to IEAXPSIM, using standard operating system conventions.
- Prepare a parameter list to pass to the simulator.
- Pass control to the simulator, using standard operating system conventions.
- Check the code returned by the simulator.
- Perform corrective action if necessary.

In addition, the exit routine may perform the following tasks:

- Load the IEAXPSIM module, using the LOAD macro instruction, before its use.
- Delete the IEAXPSIM module, using the DELETE macro instruction, after its use.
- Load the simulator, using the LOAD macro instruction, the first time it is needed.
- Delete the simulator, using the DELETE macro instruction, at the end of the job step.

Designing the Exit Routine

The following paragraphs and Figure 40 should help you design your exit routine.

The parameter list that you pass to IEAXPSIM must be pointed to by register 1 and must contain a pointer to a doubleword area into which IEAXPSIM will move the name of the simulator module to which you will pass control.

The parameter list that you pass to the simulator must be pointed to by register 1 and must contain the following:

1. A pointer to the PIE.
2. A pointer to the area containing the contents of general registers 0 through 15 at interrupt time.
3. A pointer to a work area.
4. A pointer to a byte that is nonzero if the last bit of the quotient for a DXR need not be correct.

```
         USING   EXTPRE,15
EXTPRE   STM     3,13,SIMSV+12        Save registers not in PIE
         LR      4,15
         USING   EXTPRE,4             Establish addressability
         MVC     SIMSV(12),20(1)      Registers 0-2 from PIE
         MVC     SIMSV+56(8),12(1)    Registers 14-15 from PIE
         ST      14,RET               Save return address
         ST      1,PARMB              Pointer to PIE
         LA      13,SAVESIM           Load save area address
         L       15,SIMADD
         LTR     15,15                Does SIMADD contain address?
         BNZ     TOSIM                If so, go directly to simulator
         LOAD    EP=IEAXPSIM
         LR      15,0                 Put IEAXPSIM's address in register
         LA      1,PARMA              Load pointer to doubleword
         BALR    14,15                Get simulator's address
         DELETE  EP=IEAXPSIM
         LOAD    EPLOC=SIMUL          Load simulator
         LR      15,0                 Put simulator's address in register
         ST      0,SIMADD             Save address of simulator
TOSIM    LA      1,PARMB              Parameter list address
         BALR    14,15                go to simulator
         LTR     15,15                Error or exceptional
         BZ      GOODOUT              Condition?
*HERE THE EXIT ROUTINE SHOULD DETERMINE THE ERROR OR THE
*EXCEPTIONAL CONDITION THAT OCCURRED IN SIMULATING AND
*TAKE APPROPRIATE ACTION.
         ...
         B       OUT
GOODOUT  EQU     *
*HERE THE EXIT ROUTINE SHOULD TAKE APPROPRIATE ACTION WHEN
*NO ERROR OR EXCEPTIONAL CONDITION OCCURRED DURING SIMULATION.
         ...
OUT      L       14,RET
         LM      3,13,SIMSV+12        Restore registers
         BR      14                   Return
*WHEN THE EXIT ROUTINE NO LONGER NEEDS THE SIMULATOR,
*THE ROUTINE SHOULD DELETE IT.
         ...
         DELETE  EPLOC=SIMUL
         ...
PARMA    DS      X'80',AL3(SIMUL)     Pointer to simulator name
SIMUL    DS      D                    Simulator name
PARMB    DS      F                    For pointer to PIE
         DC      A(SIMSV)             Address of register area
         DC      A(WORK)              Address of work area
         DC      X'80',A13(ZERO)      Divide adjust switch
                                       pointer
ZERO     DC      X'0'                 Adjust switch for divide
WORK     DC      50D                  Work area
SIMSV    DS      16F                  Register area
SIMADD   DC      F'0'                 Address of simulator
RET      DS      F                    Return address
SAVESIM  DS      18F                  Save area
```

Figure 40. Calling the Extended-Precision Floating-Point Simulator

The work area must be at least 30 doublewords (240 bytes) if your installation's machine model has the extended-precision floating-point facility or at least 50 doublewords (400 bytes) if it does not. The exit routine shown in Figure 40 can be used for either type machine model because its work area is 50 doublewords.

To obtain the name of the extended-precision floating-point simulator installed in your system, call the module IEAXPSIM, which returns a pointer to the name of the simulator in the doubleword that you provide. In Figure 40, the doubleword is SIMUL.

Before passing control to the simulator, you can use the LOAD macro instruction to bring the simulator into virtual storage if it is not already there. The entry point name is specified as

the name returned from IEAXPSIM. After issuing LOAD, you can pass control to the simulator, using standard calling conventions.

Upon regaining control from the simulator, the exit routine should check register 15 for one of the two return codes shown in Figure 41.

Hexadecimal Code	Meaning
00	The operation was successful.
FF	The operation was not successful, or an exceptional condition occurred.

Figure 41. Return Codes From the Extended-Precision Floating-Point Simulator

If the return code is X'FF', the exit routine determines the kind of error encountered by the simulator by examining the interruption code. Figure 42 shows the possible settings of the interruption code.

Meaning of Interruption	
The simulator found that the operation was not an extended-precision floating-point operation and returned control without further processing.	0001
Protection exception[1] [3]	0100
Addressing exception[1] [3]	0101
Specification exception[1] [2] [3]	0110
Exponent overflow exception[4]	1100
Exponent underflow exception[4]	1101
Significance exception[4]	1110
Floating-point divide[4]	1111

[1]When the simulator encounters these exceptions, it stops processing and returns control to the exit routine.
[2]An incorrect extended-precision floating-point register was specified, the third byte of the DXR macro instruction was not X'00' or a register other than 0 or 4 was specified in the R1 or R2 field of the DXR macro instruction.
[3]The error occurred during the processing of an MXD macro instruction.
[4]The error occurred during simulation.

Figure 42. Interruption Codes Returned by the Simulator

The simulator adjusts the condition code in the old PSW in the PIE (bits 34-35) to indicate the result of an AXR or SXR macro instruction. When a program interruption occurs within the simulator while fetching the argument of the MXD macro instruction, the instruction address in the PSW in the PIE is restored to its setting at operation-interruption time.

The simulator never alters the program check old PSW at location 40. Its interruption code will be an operation exception except for the MXD macro instruction, when it may be a protection, addressing, or specification exception.

The simulator should be deleted by the using program if it was obtained by the LOAD macro instruction.

If the full simulator (IEAXPALL) is loaded on a CPU that already has the extended-precision floating-point facility, no abnormal conditions result. Only the DXR macro instruction is simulated. However, the simulation of the DXR function is slower than if the IEAXPDXR were used, since the other extended-precision operations in the divide algorithm are also simulated.

If IEAXDXR is loaded on a CPU without the extended-precision floating-point facility, a 0C1 ABEND occurs when an extended-precision divide is simulated. In the simulation of the other extended-precision macro instructions, a return code of X'FF' is passed to the caller and no simulation is attempted.

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 32 operator consoles.

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.

- Only the first line of a multiple-line WTO message is passed to the user-written WTO exit routine.
- 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 leftmost 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.
- Since the hard copy log receives a copy of every message in the system, an active operator's console should be used 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 which 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 may be properly presented on, and deleted from, display devices. Each WTO macro instruction should contain one descriptor code. The descriptor code is not printed or displayed as part of the message text. If a descriptor code of 1 or 2 is coded into the WTO macro instruction, an indicator (* or @) is inserted as the first character of the message. The indicator informs the operator that he is required to take some immediate action. If a descriptor code other than 1 or 2 is coded, a blank is inserted as the first character, indicating that no immediate action is needed.

A sample WTO macro instruction is shown in Figure 43.

```
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 43. Writing to the Operator

To use the WTOR macro instruction, you code the message exactly as designated in the single-line WTOR 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 44. 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',
                    REPLY,3,ECBAD,ROUTCDE=(1,15),DESC=7
            WAIT    ECB=ECBAD
            ...
ECBAD       DC      F'0'        Event control block
REPLY       DC      C'bbb'      Answer area
```

Figure 44. 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. The master console may answer any WTOR, even if he 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.

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.

Message Deletion

If your system is using a cathode-ray tube (CRT) display as a console, unnecessary messages 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.

You can also use the DOM macro instruction to inhibit operator messages from appearing on any operator console 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 WTRO macro instruction or must be a task executing in supervisor mode, under protection key 0-7, or authorized by APF.

Part II: Macro Instructions

Introduction to Supervisor 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.

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.

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 decriptions 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 *OS/VS - DOS/VS - VM/370 Assembler Language.*

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 attempt to explain the meanings of the parameters; the parameters are explained following the table.

Figure 45 presents a sample macro instruction, TEST, and summarizes all the coding information that is available for it. The table is divided into three columns.

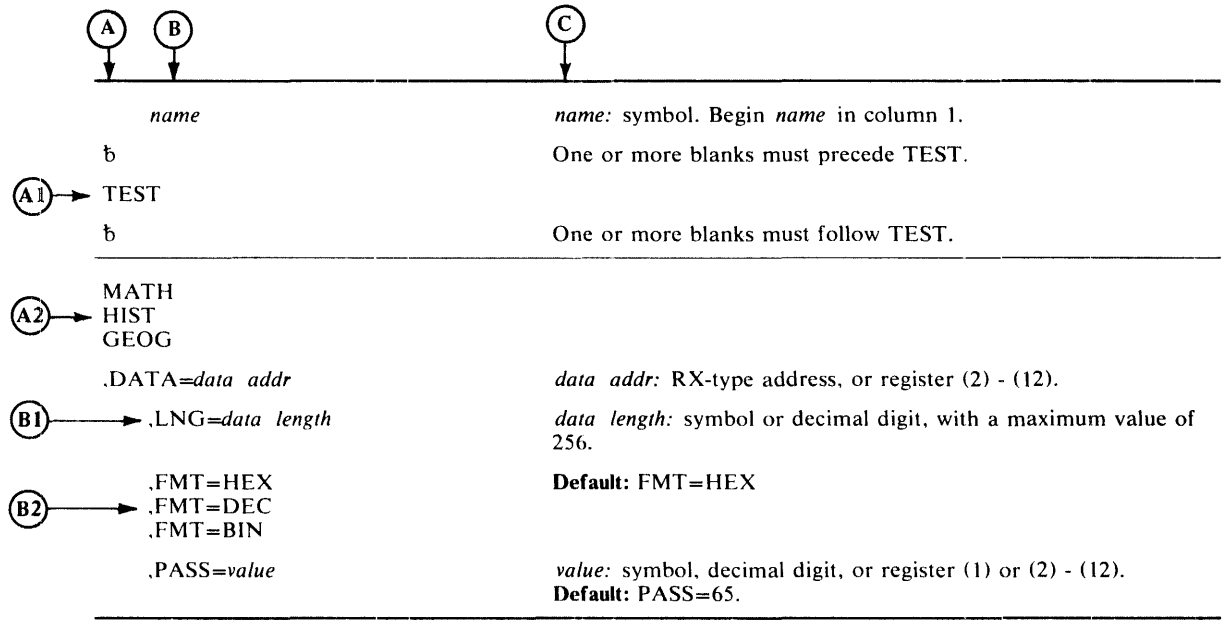


Figure 45. Sample Macro Instruction

- 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), the parameter appearing on one and only one of 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), the parameter appearing on one and only one of the lines may be coded if desired.
- The third column, (C), provides additional information for coding the macro instruction. When subsitution of a variable is required, the following classifications should be understood:

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, STEP, DUMP, or RET=USE), code the parameter exactly as shown.
- If the selected parameter is written in italics (for example, *value* or *comp code*), 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, EP=*entry point*), 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 a parameter is selected to be coded read column 3 before proceeding to the next parameter. Column 3 will often contain 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 46 shows an example of each method. Additional information on the continuation of any assembler language macro instruction is provided in the publication *OS/VS - DOS/VS - VM/370 Assembler Language.*

```
NAME1    OP1 OPERAND1,OPERAND2,OPERAND3,OPERAND4,OPERAX
             ND5,OPERAND6         THIS IS ONE WAY
NAME2    OP2 OPERAND1,OPERAND2,   THIS IS ANOTHER WAY   X
             OPERAND3,                                  X
             OPERAND4
```

Figure 46. Continuation Coding

VS1/VS2 Compatibility

This publication describes VS2 macro instructions only, However, all macro instructions and parameters defined in this publication may also be executed on a VS1 system, with the following exceptions. If these exceptions are coded, assembler errors will result.

ABEND macro instruction
- SYSTEM
- USER
- DUMPOPT=

ATTACH macro instruction
- GSPV=
- GSPL=
- SHSPV=
- SHSPL=
- SZERO=
- TASKLIB=
- STAI=
- ESTAI=
- PURGE=
- ASYNCH=
- TERM=
- RELATED=

CHAP macro instruction
- RELATED=

DELETE macro instruction
- RELATED=

DEQ macro instruction
- RELATED=

DETACH macro instruction
- STAE=
- RELATED=

DOM macro instruction
- REPLY=

ENQ macro instruction
- RELATED=

ESTAE macro instruction

FREEMAIN macro instruction
- LC
- LU
- L
- C
- VU
- EC
- EU
- RC
- RU
- LA=
- RELATED=

GETMAIN macro instruction
- RC
- RU
- LC
- LU
- RELATED=

LINK macro instruction
- ERRET=

LOAD macro instruction
- ERRET=
- RELATED=

PGLOAD macro instruction

PGOUT macro instruction

SETRP macro instruction

SNAP macro instruction
- SDATA=(LSQA,SQA,SWA)

STATUS macro instruction

STIMER macro instruction
- MICVL=

STIMER macro instruction
    MICVL=
    GMT=
    ERRET=
TIME macro instruction
    STCK
    ZONE=
    ERRET=
TTIMER macro instruction
    MIC
    ERRET=
WAIT macro instruction
    LONG=
WTO macro instruction
    multiple line message formats

Descriptions of the Macro Instructions

ABEND — Abnormally Terminate a Task

The ABEND macro instruction is used to initiate 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 following actions are taken:

- The task that was active when ABEND was issued is terminated, along with all of the subtasks of that active task.
- The completion code is posted as indicated in the completion code parameter description below.
- The end-of-task exit routine specified in the ATTACH macro instruction that created the task which issued ABEND is selected to be given control. 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
,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 below:

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.

,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 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.

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 and DCB 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 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 four ranges 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
```

ATTACH — Create a New Task

The ATTACH macro instruction causes the control program to create 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.

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 EXTR 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 EXTR parameter is not coded, the subtask is automatically removed from the system upon completion of its execution. 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.

The ATTACH macro instruction cannot be issued in a STAE exit routine. The program issuing the ATTACH macro instruction must not terminate before all of its subtasks have terminated.

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 below:

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.

,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.)

,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. (If this parameter is not coded, register 1 is not altered.)

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 to be used by the control program to indicate the termination of the new 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 the same routine is used for more than one subtask, it must be reenterable. 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 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. Ownership of each of the specified subpools is assigned to the new task. 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.)

,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 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.

,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 ESTAE 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
—ESTAE macro instruction 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 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')
```

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.

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 blanks must follow ATTACH.
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.
,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)* ,ESTAI=*(exit addr,parm 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 exceptions:

,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=*supoool 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.

Note: 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.

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.

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.

Example 1

Operation: Cause the program named in the list to be attached. Established 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,ESTAE=(EXIT1)
```

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. The CALL macro instruction cannot be used 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.

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 below:

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.

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)*,VL	*addr:* A-type address. **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 exceptions:

,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 exceptions:

,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.

Example 1

Operation: Call the entry point contained in register 15, and pass three addresses to the control program.

```
CALL     (15),(ADDR1,ADDR2,ADDR3)
```

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:* Begin *name* in column 1.
ƀ	One or more blanks must precede CHAP.
CHAP	
ƀ	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 below:

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 for a subtask of the active task. If 'S' is coded or assumed, the dispatching priority of the active task is updated.

,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 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')
```

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
```

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 below:

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 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')
```

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

DEQ removes control of one or more (maximum is 65,535) 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.

The standard form the the DEQ macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ƀ	One or more blanks must precede DEQ.
DEQ	
ƀ	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
,*var1234*	*var1234:* The preceding 4 parameters may be repeated up to 65,535 times.
)	
,RET=HAVE ,RET=NONE	**Default:** RET=NONE
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained below:

(
specifies the beginning of the resource description.

qname addr
specifies the address in virtual storage of an 8-character name. 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 and must be from 1 to 255 bytes long. 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.

,var1234
specifies that up to 65,535 resources may be specified in the DEQ macro instruction.

)
specifies the end of the resource description.

,RET=HAVE
,RET=NONE
specifies that the request for releasing the resources named in DEQ is to be honored only if the active task has been assigned control of the resources or if ENQ was executed with ECB (HAVE) or specifies an unconditional request to release all the resources (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.

,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 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')
```

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 47. 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 48.

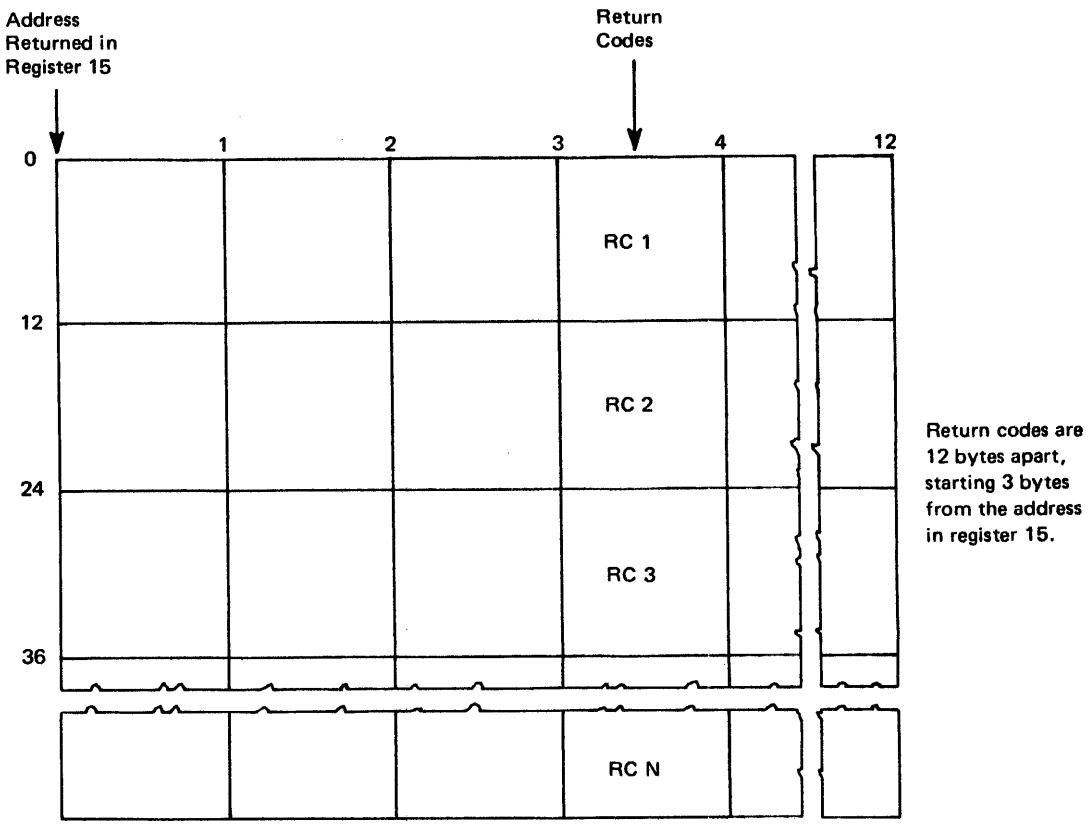


Figure 47. Return Code Area Used by DEQ

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 48. DEQ Macro Instruction Return Codes

DEQ (List Form)

Use the list form of DEQ to construct a control program 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*	*rname addr:* A-type address.
, ,*rname length*	*rname length:* symbol or decimal digit.
, ,STEP ,SYSTEM ,SYSTEMS	**Default:** STEP
var1234	*var1234:* The preceding 4 parameters may be repeated up to 65,535 times.
)	
,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 exceptions:

,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.

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
,*var1234*	*var1234:* The preceding 4 parameters may be repeated up to 65,535 times.
)	**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) or (2) - (12).

The parameters are explained under the standard form of the DEQ macro instruction, with the following exceptions:

,MF=(E ,*ctrl addr*)
specifies the execute form of the DEQ macro instruction using a remote control program parameter list.

Example 1

Operation: Release control of the resource in Example 1 of ENQ, if it has been assigned to the current TCB. 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.

```
DEQ (MAJOR4,MINOR4,3,STEP,MAJOR2,MINOR2,,SYSTEM,
    MAJOR3,MINOR3,,SYSTEMS)
```

DETACH — Detach a Subtask

The DETACH macro instruction is used to remove 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.
ƀ	One or more blanks must precede DETACH.
DETACH	
ƀ	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 below:

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.

,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 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')
```

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
```

DOM — Delete Operator Message

The DOM macro instruction is used with MCS with DIDOCS only. It is used to delete an operator message or group of messages from display on graphic consoles or to inhibit operator messages from ever appearing on any operator consoles. When a program no longer requires that a message be displayed, the DOM macro instruction should be issued 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 executed, the control program assigns an identification number to the message. The control program returns the assigned identification number (24 bits and right-justified) to the issuing program in general register 1. When display of the message is no longer needed, the DOM macro instruction is coded 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=*reg* MSGLIST=*list addr*	*reg:* register (1) or (2) - (12). *list addr:* symbol, RX-type address, or register (1) or (2) - (12).
,REPLY=YES	

The parameters are explained below:

MSG=*reg*
MSGLIST=*list addr*
specifies the message numbers of messages to be deleted.

For MSG, the 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. 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 2 through 12 is used, the macro expansion is shortened by two bytes.

,REPLY=YES
specifies that the need for a reply to a WTOR message has been eliminated. This parameter must be specified if a WTOR message is to be deleted.

Example 1

Operation: Delete an operator message whose message id is in register 1.

```
DOM MSG=(R1)
```

Example 2

Operation: Delete a list of operator messages, some of which may be WTORs.

```
DOM MSGLIST=ID2,REPLY=YES
```

DXR — Divide Extended Register

The DXR macro instruction is used to divide one extended-precision floating-point number by another. A detailed description of the division process and extended precision and rounding is given in *IBM System/370 Principles of Operation.*

To use the DXR macro instruction, you must provide a SPIE exit routine to process the program exception caused (intentionally) by execution of the DXR instruction. The SPIE exit routine is described in the section on Extended-Precision Floating-Point Simulation in the Services section of this publication.

The DXR macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede DXR.
DXR	
b	One or more blanks must follow DXR.
dividend reg	*dividend reg:* symbol or decimal digit. The only permitted registers are 0 and 4.
,divisor reg	*divisor reg:* symbol or decimal digit. The only permitted registers are 0 and 4.

The parameters are explained below:

dividend reg
specifies the register that contains the dividend. The quotient is placed in this register; the remainder is discarded.

,divisor reg
specifies the register that contains the divisor.

Example 1

Operation: Divide the extended-precision floating-point number in register 0 by the extended-precision floating-point number in register 4.

```
DXR     0,4
```

ENQ — Request Control of a Serially Reusable Resource

ENQ requests the control program to assign control of one or more (up to 65,535) serially reusable resources to the active task. If any of the resources are not available, the active task may 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 one of the programs of the same task issues a DEQ macro instruction specifying the same resource. 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 may 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. The control program does not correlate the names with the actual resource. ENQ simply coordinates access to whatever it is the names represent. The names may be given meaning restricted to a job step or across job steps. In either case, all programs for which coordination of the resource is provided must represent it by the same name.

Issuing two ENQ macro instructions 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 normal termination of a task is attempted while the task still has control of any serially reusable resources, all requests made by this task will be automatically dequeued. If resource input addresses are incorrect, the task is abnormally terminated.

The standard 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, or register (2) - (12).
,rname addr	*rname addr:* A-type address, or register (2) - (12).
, ,E ,S	**Default:** E.
, *,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.* **Default:** assembled length of *rname)i*
, *,STEP* *,SYSTEM* *,SYSTEMS*	**Default:** *STEP.*
,var12345	*var12345: The preceding 5 parameters may be repeated up to 65,535 times.*
)	
,RET=CHNG *,RET=HAVE* *,RET=TEST* *,RET=USE* *,RET=NONE*	**Default:** *RET=NONE.*
,RELATED=value	*value: any valid macro keyword specification.*

The parameters are explained below:

(
specifies the beginning of the resource description.

qname addr
specifies the address in virtual storage of an 8-character name. Every program issuing a request for a serially reusable resource must use the same *qname*, *rname*, and scope to represent the resource.

,rname addr
specifies the address in virtual storage of the name used in conjunction with *qname* to represent a single resource. The name can be qualified and must be from 1 to 255 bytes long.

,
,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.

,
,STEP
,SYSTEM
,SYSTEMS
specifies the scope of the resource used only within the job step of the issuing program (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 is specified, requests for the same *qname* and *rname* from programs of other address spaces denote the same resource; if SYSTEMS is specified, requests for the same *qname* and *rname* from programs of other address spaces in the various systems 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. Also when one resource is used by a single address space and another resource is used by several address spaces in one or more systems, the same *qname* and *rname* can be used for both.

,var12345
specifies that up to 65,535 resources may be specified in the ENQ macro instruction.

)
specifies the end of the resource description.

,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 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.

,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 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')
```

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 49. 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 50.

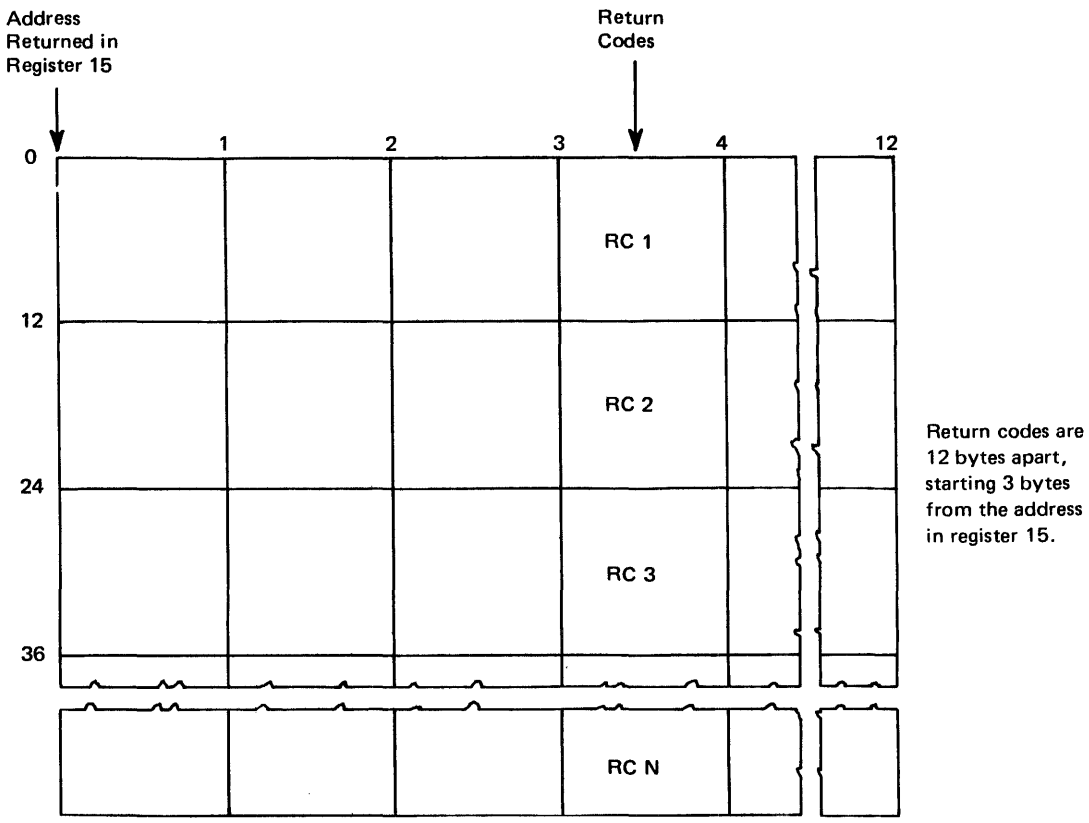


Figure 49. Return Code Area Used by ENQ

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 shared.
8	For RET=TEST, RET=USE, or RET=HAVE, a previous request for control of the same resource has been made for the same task. Task has control of resource. For RET=CHNG, the resource has not been queued. If bit 3 is on — shared control of resource; if bit 3 is off — exclusive control.
20	A previous request for control of the same resource has been made for the same task. Task does not have control of resource.

Figure 50. ENQ Return Codes

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
,var12345	*var12345:* The preceding 5 parameters may be repeated up to 65,535 times.
)	
,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 exceptions:

,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.
ƀ	One or more blanks must precede ENQ.
ENQ	
ƀ	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
,*var12345*	*var12345:* The preceding 5 parameters may be repeated up to 65,535 times.
)	**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) or (2) - (12).

The parameters are explained under the standard form of the ENQ macro instruction, with the following exceptions:

,MF=(E ,*ctrl addr*)
specifies the execute form of the ENQ macro instruction using a remote control program parameter list.

Example 1

Operation: Request 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 third resource is 8 characters.

```
ENQ     (MAJOR4,MINOR4,E,,,MAJOR2,MINOR2,,,SYSTEM,
        MAJOR3,MINOR3,E,8,SYSTEMS)
```

ESTAE — Extended STAE

The ESTAE macro instruction is used to extend the recovery capability facilities of the STAE (Specify Task Abnormal Exit) macro instruction. Issuance of the STAE or ESTAE macro instruction or ATTACH with the STAI or ESTAI 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 modes.

ESTAE provides the increased capabilities over STAE to allow ESTAE exits to be scheduled for clean-up processing under certain instances for which STAE exits did not get control, and to default parameters to the most commonly used options.

Note: The STAE macro instruction is available for compatibility with Release 1 of VS2 and with MVT and MFT, and is described in *OS/VS2 System Programming Library: Job Management, Supervisor, and TSO.* However, it is recommended that ESTAE be used.

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 below:

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 canceled.

,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 canceled (NO) or will not be canceled (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.

Notes: If any IBM-supplied access method, except EXCP, is being used, the PURGE=NONE option is recommended. If this is done, 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.

ISAM Notes: If ISAM is being used and PURGE=HALT is specified or PURGE=QUIESCE is specified but I/O is not restored:

- Only the input/output event on which the purge is done will be posted. Subsequent event control blocks (ECBs) will not be posted.
- The ISAM check routine will treat purges I/O as normal I/O.
- Part of the data set may be destroyed if the data set is 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 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')
```

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 nonexistant, not owned by the user's RB, or is not an ESTAE exit.
0C	Cancel (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.

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 exceptions:

,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 ,OV	
,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 exceptions:

,MF=(E ,*ctrl addr*)
specifies the execute form of the ESTAE macro instruction using a remote control program parameter list.

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, contain 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))
```

EVENTS — Wait for One or More Events to Complete

The EVENTS macro instruction is a functional specialization of the WAIT ECBLIST= macro facility with the advantages of notifying the program that events 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.

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* ENTRIES=DEL,TABLE=*table address* TABLE=*table address*	*n:* variable. decimal digit 1-32,767. *table address:* symbol, RX-type address, or register (2)-(12). **Note:** If ENTRIES=*n* or ENTRIES=DEL,TABLE=*table address* is specified, no other parameter should be specified.
,WAIT=NO ,WAIT=YES	**Default:** None.
,ECB=*ecb address* ,LAST=*last address*	*ecb address:* symbol, RX-type address, or register (2)-(12). *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 below:

ENTRIES = n

n is a decimal number from 1 to 32,767 which specifies the maximum number of completed ECB addresses that can be processed in an EVENTS table concurrently.

Note: When this parameter is specified no other parameter should be specified.

ENTRIES = DEL,TABLE = table address

specifies 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.

Note: When this parameter is specified no other parameter should be specified.

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.

,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 initializes the ECB, thus identifying it as being eligible for POSTing. The ECB must be initialized only after it is eligible for POSTing.

Note:

- Register 1 should not be specified for the ECB address.
- This parameter may not be specified with the LAST= parameter.
- 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.

Note:

- Register 1 should not be specified for the LAST address.
- This parameter should not be specified with the ECB= parameter.

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 51 below:

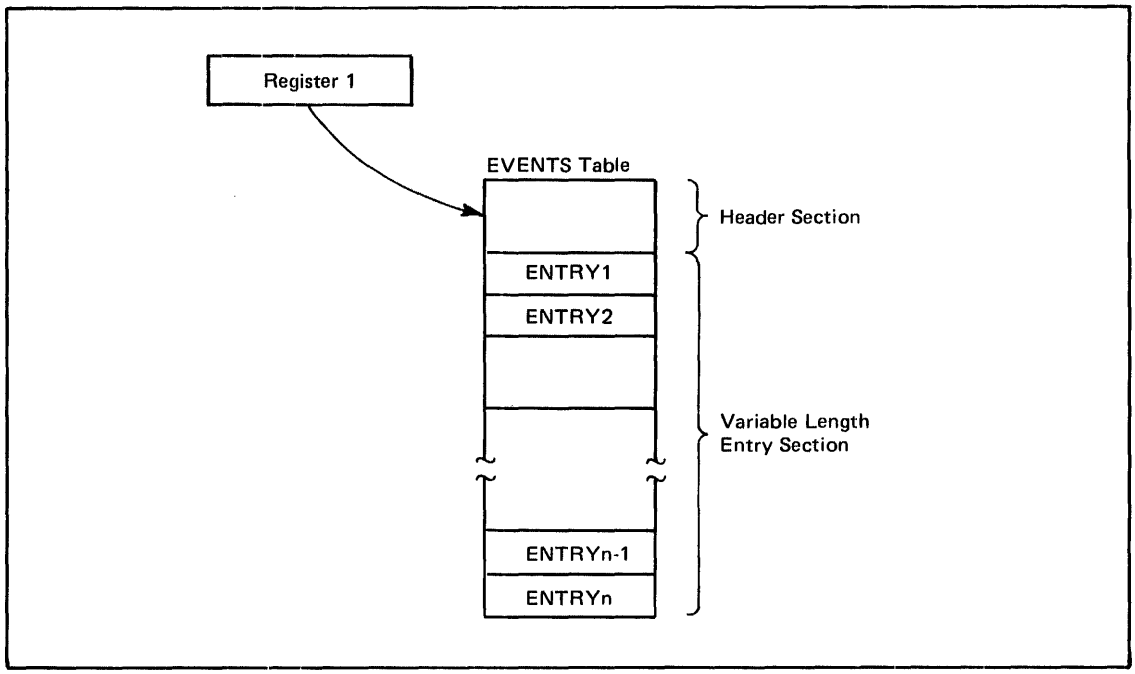


Figure 51. Creating a Table

- 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 set 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:

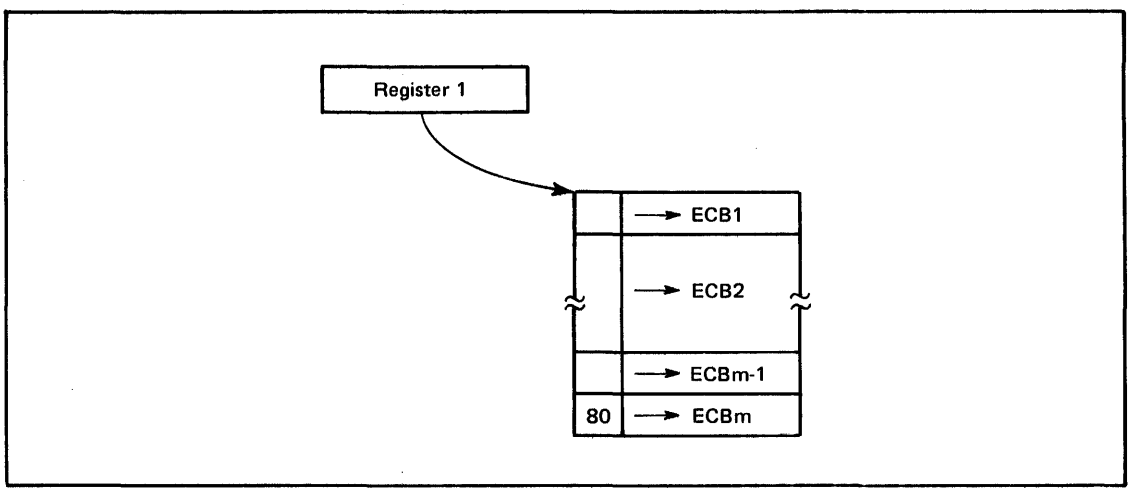


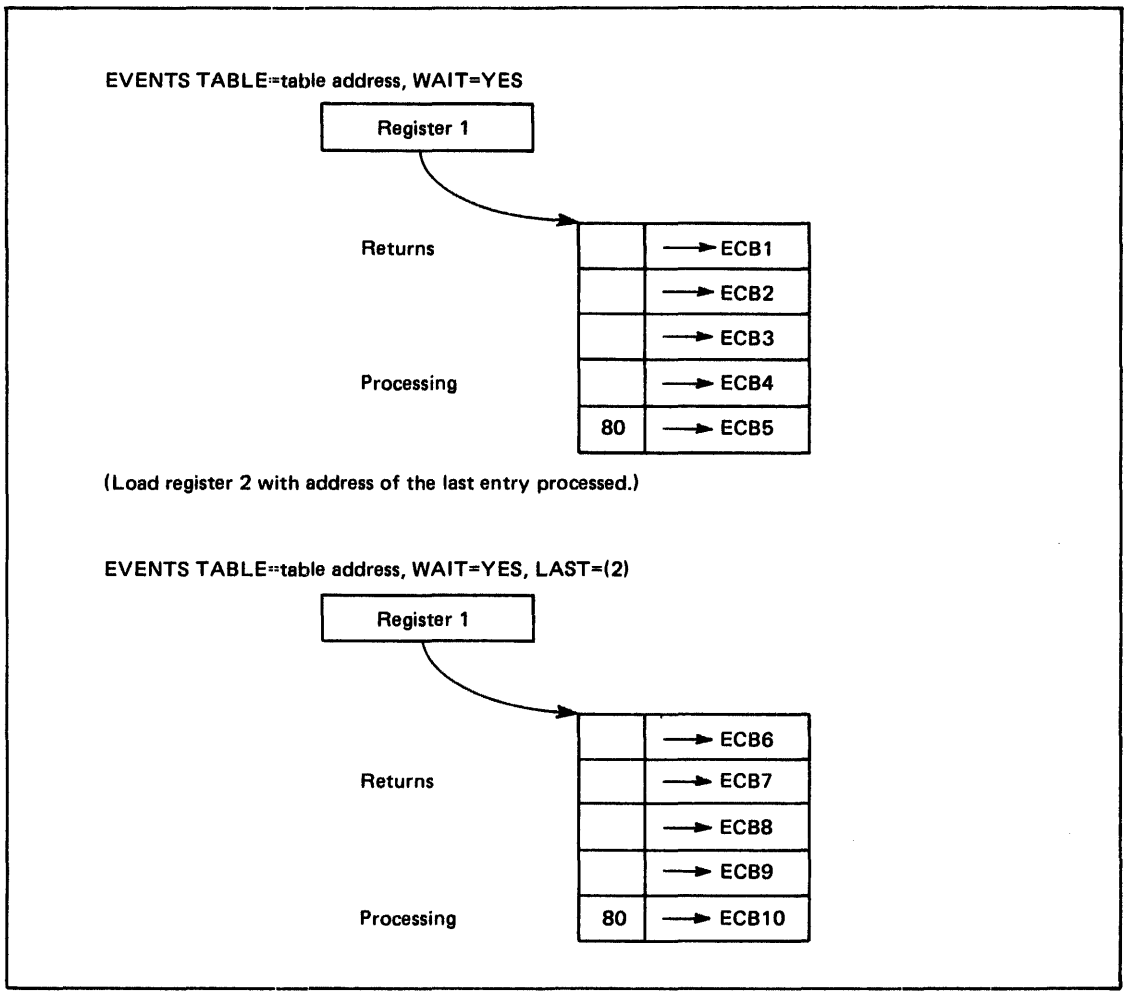
Figure 52. Parameter List Format

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 parmeter 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 below, ECBs 6 through 10 were posted to the parameter list while the user was processing 1 through 5.



This figure demonstrates processing one event at a time.

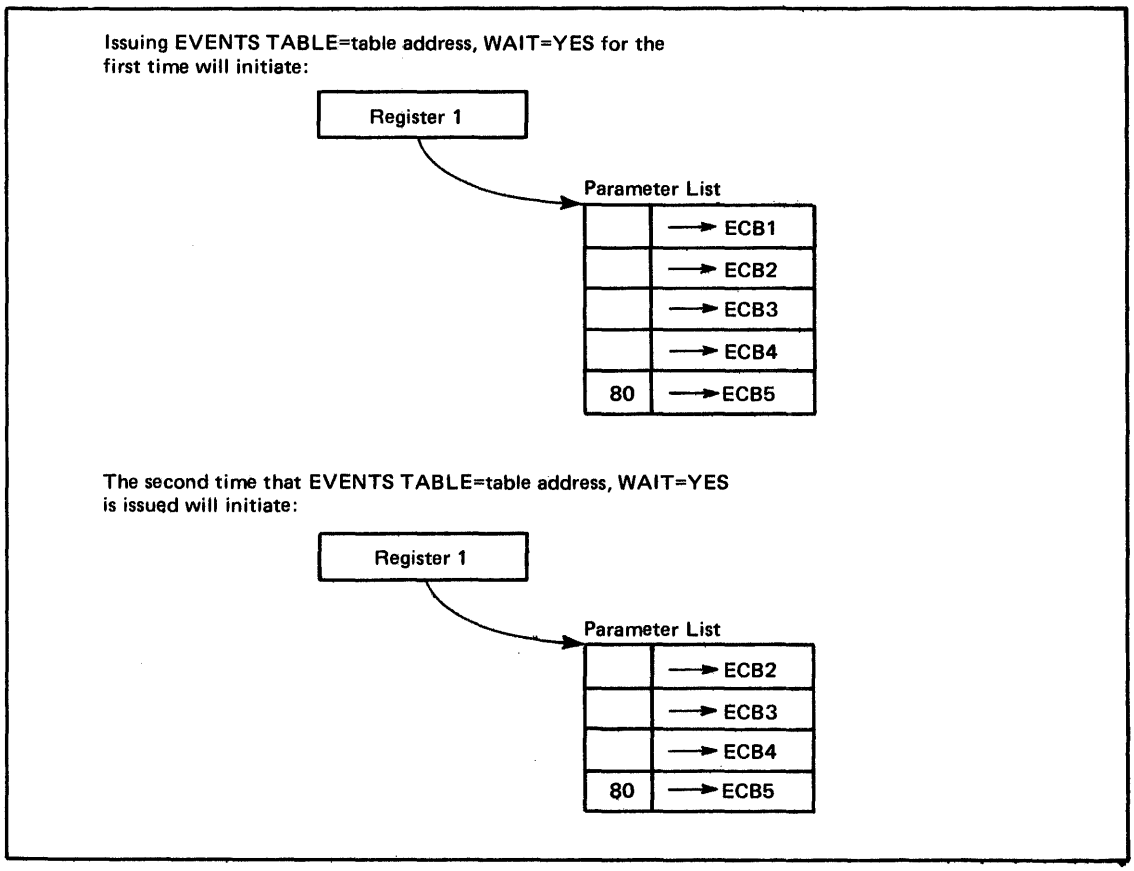


Figure 53. Processing One Event At A Time

Example 1

The following shows total processing via EVENTS.

EVENTS & 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=TABLEADDR,ECB=(3),WAIT=YES
                     WRITE       DECBRW,K,MF=E
                     LA          3,DECBRW
         RETEST      EVENTS      TABLE=TABLEADDR,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 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:** If the forms RC,SP=*subpool nmbr* or RU,SP=*subpool nmbr* or R,SP=*subpool nmbr* are specified, no other parameters may be specified.
,A=*addr*	*addr:* A-type address, or register (2) - (12).
,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 below:

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 and LU and L indicates conditional (LC) and unconditional (LU and L) list requests, and specifies 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 and VU and V indicates conditional (VC) and unconditional (VU and V) variable requests, and specifies 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 and EU and E indicates conditional (EC) and unconditional (EU and E) element requests, and specifies 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 and RU and R indicates conditional (RC) and unconditional (RU and R) register requests, and specifies release of single areas of virtual storage from the subpool indicated, or specifies 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.

Note: 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.

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. If the words are within an area to be released, they must be completely within the area and must not begin in the first two words of the first area. 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. If the words are within an area to be released, they must be conpletely within the area and must not begin in the first two words of the first area. 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, VS, 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. The specification of register 1 saves two bytes in the macro expansion.

,SP=subpool nmbr

specifies the subpool number of the virtual storage area to be released. The subpool number can be between 0 and 127. If the SP parameter is optional and is omitted, subpool 0 is assumed. If the SP parameter must be coded, it specifies the number of the subpool to be released, and the valid range is 1 through 127. Subpool 0 cannot be released. If R is coded, SP=(0) can be designated, in which case the subpool number must be previously loaded into the high-order byte of register 0; the three low-order bytes must be set to 0.

,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 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')
```

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.

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 specification.
,MF=L	

The parameters are explained under the standard form of the FREEMAIN macro instruction, with the following exceptions:

,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	*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.
,LA=*length addr* ,LV=*length value*	
,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 specification.
,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 FREEMAIN macro instruction, with the following exceptions:

,MF=(E ,*ctrl prog*)
specifies the execute form of the FREEMAIN macro instruction using a remote control program parameter list.

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,A=(2)
```

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
```

GETMAIN — Allocate Virtual Storage

The GETMAIN macro instruction requests the control program to allocate 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 total of the lengths specified must not exceed the length available when the task assigned ownership terminates, or through the use of the FREEMAIN macro instructions.

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).
,SP=*subpool nmbr*	*subpool nmbr:* symbol, decimal digit 0-127, or register (2) - (12). **Note:** If R,LV=(0) is specified above, SP may not be specified.
,BNDRY=DBLWD ,BNDRY=PAGE	**Default:** BNDRY=DBLWD **Note:** This parameter may not be specified with R above.
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained below:

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*

specifies the type of GETMAIN request:

LC and LU indicates conditional (LC) and unconditional (LU) list requests, and specifies 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 indicates conditional (VC) and unconditional (VU) variable requests, and specifies 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 indicates conditional (EC) and unconditional (EU) element requests, and specifies requests for single areas of virtual storage. The length of the single virtual storage area is indicated in the LV parameter. The address of the allocated virtual storage area is returned at the address indicated in the A paramater.

RC and RU and R indicates conditional (RC) and unconditional (RU and R) register requests, and specifies requests for single areas of virtual storage. The length of the single virtual area is indicated in the LV parameter. The address of the allocated virtual storage area is returned in register 1. (R generates the original SVC 10 calling sequence, whereas RU generates a new SVC 120 and associated parameter format.)

Note: 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.

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 ligh-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 contain 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 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.

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. If this parameter is omitted, subpool 0 is assumed.

,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.

,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 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')
```

When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Virtual storage requested was allocated.
04	No virtual storage was allocated.

GETMAIN (List Form)

Use the list form of the GETMAIN macro instruction to construct a control program paramater 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 not be specified with EC or EU above. **Note:** LV may not be specified with LC, LU, VC, 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 specification.
,MF=L	

The parameters are explained under the standard form of the GETMAIN macro instruction, with the following exceptions:

,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 not be specified with EC or EU above. **Note:** LV may not 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 specification.
,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 exceptions:

,MF=(E ,*ctrl prog*)
specifies the execute form of the GETMAIN macro instruction using a remote control program parameter list.

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 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 storage is not available, the task will be abnormally terminated.

```
GETMAIN     EU,LV=48,A=AREAADDR
     .
     .
AREAADDR    DS      F
```

IDENTIFY — Add an Entry Name

The IDENTIFY macro instruction is used to add 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 WTO 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.

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* EPLOC=*entry name addr*	*entry name:* symbol *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 below:

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 addresss of the entry name 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	Request was previously issued using the same entry name but a different address; request was isnored.
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.

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 is used to pass 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. The load module containing the program is brought into virtual storage if a useable copy is not available.

The linkage relationship established is the same as that created by a BAL instruction; control is returned to the instruction following the LINK macro instruction after execution of the called program. 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.

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).

The parameters are explained below:

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.

,DCB=*dcb addr*

specifies the address of the 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 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.

,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.)

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 the routine to be given control when an error condition other than input parameter errors is detected.

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.
,SF=L	

The parameters are explained under the standard from of the LINK macro instruction, with the following exceptions·

,SF = L
specifies the list form of the LINK macro instruction.

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:* A-type address.
,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 ,*prob 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.

Example 1

Operation: Pass control to a specified entry name (PGMLKRUS) in another load module. Let the system find the module form available libraries.

```
LINK     EP=PGMLKRUS
```

LOAD — Bring a Load Module into Virtual Storage

The LOAD macro instruction is used to bring the load module containing the specified entry name into virtual storage, if a usable copy is not available in virtual storage.

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 reducted 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.

The entry name in the load module must be a member name or an alias in a directory of a partitioned data set. 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* EPLOC=*entry name addr* DE=*list entry addr*	*entry name:* symbol. *entry name addr:* RX-type address or register (0) or (2) - (12). *list entry addr:* RX-type address, or register (2) - (12).
,DCB=*dcb addr*	*dcb addr:* RX-type address, or register (1) or (2) - (12).
,ERRET=*err rtn addr*	*err rtn addr:* RX-type address, or register (2) - (12).
,RELATED=*value*	*value:* any valid macro keyword specification.

The parameters are explained below:

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.

,DCB=*dcb addr*
specifies the address of the 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 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.

,ERRET=*err rtn addr*
specifies the address of the routine to be given control when an error condition other than input parameter errors is detected.

,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 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')
```

Example

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
```

PGLOAD — Load Virtual Storage Areas into Real Storage

The PGLOAD macro instruction is used to load specified virtual storage areas into real storage in anticipation of future needs. That is, PGLOAD is essentially a page-ahead function. 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 below:

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.
04	Operation abnormally terminated. Operation incomplete because of invalid address in virtual subarea list entry; ECB posted complete.
08	Operation proceeding; ECB will be posted when all page-ins are complete.
10	Operation abnormally terminated. 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.
4 — Operation incomplete because of invalid address in VSL entry.

If the return code issued is 8, the ECB is posted asynchronously when paging I/O has completed, with code

0 — Operation completed successfully.
4 — Operation incomplete because of paging error; requesting TCB will be abnormally terminated.

Incompatible Parameters

The following parameters were valid in Release 1 of OS/VS2, but are not supported in Release 2:

ECBIND=*address*
will probably cause errors.

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) or (2) - (12).
,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.

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
```

PGOUT — Page Out Virtual Storage Areas from Real Storage

The PGOUT macro instruction is used to initiate page-out operations for specified virtual storage areas that are in real storage. 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). **Default:** *start addr* + 1
,KEEPREL=N ,KEEPREL=Y	**Default:** KEEPREL=N

The parameters are explained below:

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.
04	Operation abnormally terminated. Operation incomplete because of invalid address in virtual subarea list entry.
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, or 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 or ECB address invalid.

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.

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
```

PGRLSE — Release Virtual Storage Contents

The PGRLSE macro instruction is used to release to the system all real storage and auxiliary storage associated with specified virtual storage areas. 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 contents of the space is nullified. 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 below:

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.

PGRLSE (List Form)

The list form of the PGRLSE macro instruction is used to construct 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 exceptions:

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 exceptions:

MF=(E,*ctrl addr*)
specifies the execute form of the PGRLSE macro instruction using a remote control program parameter list.

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
```

POST — Signal Event Completion

Use the POST macro instruction to have the specified ECB (event control block) set 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. The bits in the ECB are set as follows:

Bit 0 of the specified ECB is set to 0 (wait bit).
Bit 1 is set to 1 (complete bit).
Bits 8 through 31 are set to the specified completion code.

The POST macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede POST.
POST	
b	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 - 2^{24}-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 a fullword on a fullword boundary containing the address of an 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 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')
```

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. 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 below:

(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. A byte containing all 1's is placed in the high-order byte of word 4 of the save area 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.)

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 below:

(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 if *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 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, but is returned to the instruction following the SEGLD macro instruction. Processing is overlapped with the loading of the segment. Refer to the OS/VS Linkage Editor and Loader for details on overlay.

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 below:

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 *OS/VS Linkage Editor and Loader* for details on overlay operations. The SEGWT macro instruction cannot be used in an asynchronous exit routine.

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 below:

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 is used to indicate the various requests that a recovery exit may make. It may be used 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 which 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.
,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 below:

,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 may be created by using the list form of the SNAP macro instruction, or a compatible list may be created. The TCB and DCB 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 the error; the DCB used will be one created by the control program and using as a DDNAME either SYSABEND or SYSUDUMP.

,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 the registers. 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 SDAWSRSV 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),COMPCOD=(X'00A',USER)
```

Example 2

Operation: Retry using the address specified at location 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,REGS=(14,12),RETADDR=X
```

SNAP — Dump Virtual Storage and Continue

The SNAP macro instruction is used to obtain a dump of some or all of the storage assigned to the current job step. Some or all of the control program fields can also be dumped.

You must provide a data control block and issue an OPEN macro instruction for the data set before an SNAP macro instructions are issued. The DCB macro instruction *must* contain the following parameters:

```
DSORG=PS,RECFM=VBA,MACRF=(W),BLKSIZE=nnn,LRECL=125,
and DDNAME=any name but SYSABEND or SYSUDUMP
```

BLKSIZE must be either 882 or 1632. A SNAP data set that is opened in a problem program that will be processed by the system loader should be closed by the problem program.

The data set containing the dump can reside on any device supported by BSAM (basic sequential access method). The dump is placed in the data set described by the DD statement the user provides. If a printer is selected, the dump is printed immediately; if a direct access or tape device is designated, a separate job must be scheduled to obtain a listing of the dump.

Sufficient unused storage must be available in the area assigned to the job step to hold the control program dump routine and, if not already in storage, the BSAM data management routines.

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 only one code is specified, the parentheses need not be coded. NUC CB SQA Q LSQA TRT SWA
,PDATA=ALL ,PDATA=*(prob data code)*	*prob data code:* any combination of the following, separated by commas. If only one code is specified, the parentheses need not be coded. PSW REGS SA or SAH JPA or LPA or ALLPA SPLS
,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)*

The parameters are explained below:

DCB=*dcb addr*
specifies the address of a previously opened data control block for the data set that is to contain the dump.

,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.

,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 following fields.
NUC — All of the control program nucleus except the trace table.
SQA — The system queue area.
LSQA — The local system queue area.
SWA — The scheduler work area related to the task.
CB — The control blocks for the task.
Q — The enqueue control blocks for the task.
TRT — The GTF trace table.

If a dump occurs in a GTF address space, no attempt will be made to include trace information.

,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 and general registers when the SNAP or ABEND macro instruction was issued.
SA — Save area linkage information and a back trace through save areas.
SAH — Save area linkage information.
JPA — Contents of job pack area.
LPA — Contents of link pack area.
ALLPA — Contents of job pack area and link pack area.
SPLS — All virtual storage subpools (0-127).

,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. The areas between the starting and ending addresses are dumped one full word at a time. If the addresses are not fullword multiples, they are rounded up or down to fullwords. The list must begin on a fullword boundary. The high order bit of the fullword containing the last ending address of the list must be set to 1.

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, or sufficient storage was not available.
0C	Data control block type (DSORG, RECFM, MACRF, BLKSIZE, or LRECL) was incorrect.

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 only one code is specified, the parentheses need not be coded. NUC CB SQA Q LSQA TRT SWA
,PDATA=ALL ,PDATA=*(prob data code)*	*prob data code:* any combination of the following, separated by commas. If only one code is specified, the parentheses need not be coded. PSW REGS SA or SAH JPA or LPA or ALLPA SPLS
,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)*
,MF=L	

The parameters are explained under the standard form of the SNAP macro instruction, with the following exceptions:

,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 one or more of the SDATA, PDATA, LIST parameters, the bit settings 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 only one code is specified, the parentheses need not be coded. NUC CB SQA Q LSQA TRT SWA
,PDATA=ALL ,PDATA=*(prob data code)*	*prob data code:* any conbination of the following, separated by commas. If only one code is specified, the parentheses need not be coded. PSW REGS SA or SAH JPA or LPA or ALLPA SPLS
,STORAGE=*(strt addr,end addr)* ,LIST=*list addr*	*strt addr:* RX-type address, or register (2) - (12). *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)*
,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.

,MF = (E ,*ctrl addr)*
specifies the execute form of the SNAP macro instruction using a remote control program parameter list.

Example 1

Operation: Dump the storage ranges pointed to by register 9, and dump all PDATA and SDATA options.

```
SNAP    DCB=(8),TCB=(16),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=(0),ID=4,LIST(9),SDATA=(TRT,Q)
```

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.

The effect of each SPIE macro instruction issued in performance of a task supersedes the effect of the previous SPIE issued in performance of the same task. The specified exit routine is given control when one of the specified program interruptions occurs in any program of the task.

The SPIE macro instruction can be issued by any subtask of the task; the resulting environment exists for the entire subtask.

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.

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:** ((2,4),6,8,(10,13),15)

The parameters are explained below:

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

Note: If a specified program interruption type is maskable, the corresponding bit is set to 1. Interruption types not specified above are handled by the control program.

Note: 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 exmaple, (4,8) indicates interruption types 4 and 8; ((4,8)) indicates interruption types 4 through 8.

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.
ƀ	One or more blanks must precede SPIE.
SPIE	
ƀ	One or more blanks must follow SPIE.
exit addr	*exit addr:* A-type address.
,(interrupts)	*interrupts:* decimal digits 1-15, expressed as **single values:** (2,3,4,7,8,9,10) **ranges of values:** ((2,4),(7,10)) **combinations:** ((2,4),6,8,(10,13),15)
,MF=L	

The parameters are explained under the standard form of the SPIE macro instruction, with the following exceptions:

,MF = L
specifies the list form of the SPIE macro instruction.

SPIE (Execute Form)

A remote control program parameter list (program interruptions control area) is used in, and can be modified by, the execute form of the SPIE macro instruction. The PICA (program interruptions control area) 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	*exit addr:* RX-type address, or register (2) - (12).
,(interrupts)	*interrupts:* decimal digits 1-15, expresses as **single values:** (2,3,4,7,8,9,10) **ranges of values:** ((2,4),(7,10)) **combinations:** ((2,4),6,8,(10,13),15)
,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 exceptions:

,MF = (E,*ctrl addr)*
specifies the execute form of the SPIE macro instruction using a remote control program parameter list.

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))
```

STATUS — Change Subtask Status

The STATUS macro instruction lets the programmer change the dispatchability status of one or all of his program's subtasks. For example, the STATUS macro instruction can be used 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* ,SYNCH	*tcb addr:* RX-type address, or register (2) - (12).
,RELATED=*value*	*value:* Any valid macro keyword specification.

The parameters are explained below:

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 which 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 CPU. A method of preventing this possibility is by issuing the STATUS macro with the STOP and SYNCH parameters.

,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.

,SYNCH
specifies that the STOP function effects all the subtasks of the caller. If any of the subtasks are deferring STOPs when the request is issued, the caller is placed in a WAIT condition. The issuer is taken out of the wait state when all deferred STOPs are complete.

Note: When using the STOP,SYNCH parameters extreme caution should be exercised to prevent interlocks within an address space.

The STOP-SYNCH function is performed by processing each of the subtasks of the issuer and either setting it non-dispatchable or marking it with a "stop pending" indicator (the latter occurs when stops are currently being deferred for a subtask). When at least one stop has

been deferred, the issuer is placed in a wait condition until all "stop pendings" have taken effect. Interlocks occur when a subtask, that has stops deferred, requires a resource or function that a non-dispatchable subtask owns. Thus, when using STATUS with STOP,SYNCH parameters, an interlock can occur when the following conditions occur simultaneously:

- One subtask (that has stops deferred) is waiting for a rousource that will not be available until the STOP,SYNCH issuer starts the task that owns the resource.
- The STOP,SYNCH issuer is waiting for all subtasks to become non-dispatchable.

One method of preventing this type of interlock is to establish a timer exit, via the STIMER macro, before specifying STOP with the SYNCH parameter. Then if the interlock occurs, the issuer's timer exit will get control and the subtask(s) can be restarted.

,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 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')
```

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 is used to set a programmer timer to a specified time interval (less than 24 hours) or to an interval that will expire at a specified time of day. An optional timer completion routine is given control when the time interval expires; if no 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. A second STIMER macro instruction issued before the first time interval expires overrides the first interval and exit routine.

The time interval may be a 'real-time interval' (measured continuously in real time), 'task time interval' (measured only while the task is in execution.) 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.

The STIMER macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede STIMER.
STIMER	
b	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 below:

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):

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 must be in virtual storage when it is required. 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.

,BINTVL=*stor addr*
,DINTVL=*stor addr*
,GMT=*stor addr*
,MICVL=*stor addr*
,TOD=*stor addr*
,TUINTVL=*stor addr*

specifies that the time be returned:

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 unpacked 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 unpacked 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 unpacked 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).

,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; if this parameter is omitted, the STIMER function would be abnormally terminated.

Notes:

- The time interval specified by an STIMER macro instruction has no relation to the time interval specified in an EXEC statement.
- If WAIT is specified in a system running a single task, no production work is performed while the time interval is in effect. Notify the system operator not to cancel the job.
- If the optional exit routine address and WAIT are not specified, no indication of completion of the time interval is provided.
- The TTIMER macro instruction provides a facility for determining the remaining time interval associated with STIMER.

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 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
```

TIME — Provide Time and Date

The TIME macro instruction causes the control program to return either the local time of day and date or the Greenwich mean time of day and date. The time of day and date are only as accurate as the corresponding information entered by the operator, and the system response time.

The date is returned in register 1 as packed decimal digits of the form

00YYDDDC, where:	YY	is the last two digits of the year
	DDD	is the day of the year
	C	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 is returned in 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 below:

DEC
BIN
TU
MIC *,stor addr*
STCK *,stor addr*
specifies that the time of day be returned:

For DEC, the time of day is returned in register 0 as packed 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 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.

,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. If this parameter is omitted, the TIME function would be abnormally terminated.

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 the time-of-day clock is unusable in a uniprocessing system or if both time-of-day clocks are unusable in a multiprocessing system.

```
TIME    STCK,(2),ERRET=TIMEERR
```

TTIMER — Test Interval Timer

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. TTIMER can also be used to cancel the remaining time interval.

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.

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 below:

CANCEL
specifies that the remaining time interval and exit routine, if any, are to be 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. If this parameter is omitted, the TTIMER function would be abnormally terminated.

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 is used to inform 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 below:

event nmbr,
specifies the number of events waiting to occur.

ECB=*ecb addr*
ECBLIST=*ecb list addr*
specifies the address of a fullword on a fullword boundary containing the address of an ECB 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 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 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')
```

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 you task via the WAIT macro instruction until a specified number of these activities has 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
```

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)
```

WAITR — Wait for One or More Events

The WAITR macro instruction is executed in exactly the same manner as the WAIT macro instruction. Although the LONG option is not available on the WAITR macro instruction, WAITR is interpreted as having a long wait.

Note: The WAITR macro instruction is available for compatibility with MFT.

The WAITR macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
b	One or more blanks must precede WAITR.
WAITR	
b	One or more blanks must follow WAITR.
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).

The parameters are explained below:

event nmbr,
specifies the number of events waiting to occur.

ECB=*ecb addr*
ECBLIST=*ecb list addr*
specifies the address of an ECB 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.

Example 1

Operation: Wait for one event to occur (with a default count).

```
WAITR   ECB=WAITECB
```

Example 2

Operation: Wait for 2 event to occur.

```
        WAITR   2,ECBLIST=LISTECBS
        .
        .
LISTECBS    DC  A(ECB1)
            DC  A(ECB2)
            DC  X'80'
            DC  AL3(ECB3)
```

WTL — Write To Log

The WTL macro instruction causes a message to be written 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.

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 parameters are explained below:

'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.

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 parameters are explained under the standard form of the WTL macro instruction, with the following exceptions:

,MF = L
specifies the list form of the WTL macro instruction.

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).

The parameters are explained under the standard form of the WTL macro instruction, with the following exceptions:

MF=(E ,*ctrl addr)*
specifies the execute form of the WTL macro instruction. This form uses a remote control program parameter list.

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))
```

WTO — Write to Operator

The WTO macro instruction causes a message to be written to one or more operator consoles.

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 124 characters. The permissable *line types* and *text* lengths are shown below: line type VS2 text C 34 char L 70 char D 70 char DE 70 char E ------ **Default:** D Up to 10 occurrences of the second and/or third formats may be coded.
,ROUTCDE=*(route code)*	*route code:* decimal digit from 1 to 16. The *route 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 below:

'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 not appear on the console. It can include any character that can be used in a character (C-type) DC instruction, except the New Line control character (punch combination 11-9-5). 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 may 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 will appear 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 (the apostrophes do not appear on the operator console). Any character valid in a C-type DC instruction may be coded except a New Line control character. The maximum number of characters depends on which line type is specified.

Note: The leftmost 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=*route code*
specifies the routing code(s) to be assigned to the message.

The routing codes are:

1	Master Console Action	
2	Master Console Information	
3	Tape Pool	
4	Direct Access Pool	
5	Tape Library	
6	Disk Library	
7	Unit Record Pool	
8	Teleprocessing Control	
9	System Security	
10	System Error/Maintenance	
11	Programmer Information	
12	Emulators	
13	Reserved for customer use	
14	Reserved for customer use	
15	Reserved for customer use	
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.

The descriptor codes are:

1	System Failure	
2	Immediate Action Required	
3	Eventual Action Required	
4	System Status	
5	Immediate Command Response	
6		Job Status
7		Application Program/Processor
8		Out-of-Line Message
9		Operator Request
10		Dynamic Status Displays
11-16		Reserved for future use

Note: All WTO messages with descriptor codes of 1 or 2 are action messages. An asterisk is printed before the first character of an action message to indicate a need for operator action.

If both the ROUTCDE and DESC parameters are omitted and the message is not a MLWTO message, the routing code specified in the OLDWTOR parameter of the system generation SCHEDULR macro instruction is assigned; if the OLDWTOR parameter is omitted, no routing code is assigned. Routing codes should be used with MLWTO messages. If DESC is specified with no ROUTCDE a default of zero routing code will be generated.

When control is returned, general register 1 contains the identification number (24 bits and right-justified) assigned to the message.

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.
12	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.
16	Request specified routing code 11 (WTP). Request is ignored for routing code 11 but is processed for other routing codes if specified.
20	MLWTO request to hard copy only. Request is ignored.

Note: No return codes are issued by the WTO service routine if the MLWTO feature is not used.

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')* *('text',line type)*	*msg:* Up to 124 characters The permissable *line types* and *text* lengths are shown below: line type VS2 text C 34 char L 70 char D 70 char DE 70 char E ---- **Default:** D Up to 10 occurrences of the second and/or third formats may be coded.
,ROUTCDE=*(route code)*	*route code:* decimal digit from 1 to 16. The *route 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 exceptions:

,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 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).

The parameters are explained under the standard form of the WTO macro instruction, with the following exceptions:

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 WTO message to all active consoles.

```
WTO     'NDP00005   ENDED',ROUTCDE=
                    (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)
```

Example 2

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 causes a message requiring a reply to be written to one or more operator consoles and the system log. The macro instruction also provides the information required by the control program to return the reply to the issuing program.

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 121 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=*(route code)*	*route code:* decimal digit from 1 to 16. The *route code* is one or more codes, separated by commas.

The parameters are explained below:

'msg'
specifies the message to be written to the operator's console. The message must be enclosed in apostrophes, which will not appear on the console. It can include any character that can be used in a character (C-type) DC instruction, except the New Line control character (punch combination 11-9-5). 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.

,ROUTCDE=*(route 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.

Ignored Parameters

The parameter DESC=*(desc code)* is meaningless if coded in Release 2 of OS/VS2 since all WTOR messages are assigned descriptor codes of 7 (application program/processor).

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 121 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=*(route code)*	*route code:* decimal digit from 1 to 16. The *route 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 exceptions:

,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 exceptions:

,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.

Example 1

Operation: Write a WTOR message to all active consoles.

```
WTOR    'THIS IS WTOR NUMBER 001',REPLY, 18,ECB1,
        ROUTCDE=(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)
```

XCTL — Pass Control to a Program in Another Load Module

The XCTL macro instruction causes control to be passed 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. The load module containing the entry name is brought into storage if a usable copy is not available. The storage occupied by the load module that issued the XCTL is eligible for reassignment by the control program is no other requirement exists for that load module. The program executing the XCTL macro instruction is logically removed from the active task, and the program gaining control is established 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 lowered 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 conditions 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.

The standard form of the XCTL macro instruction is written as follows:

name	*name:* symbol. Begin *name* in column 1.
ъ	One or more blanks must precede XCTL.
XCTL	
ъ	One or more blanks must follow XCTL.
(reg1), *(reg1,reg2),*	*reg1* and *reg2:* decimal digits or A-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:* 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).

The parameters are explained below:

(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.

,DCB=*dcb addr*

specifies the address of the 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 D 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 be 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.

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.
SF=L	

The parameters are explained under the standard form of the XCTL macro instruction, with the following exceptions:

SF=L
specifies the list form of the XCTL macro instruction.

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 or registers (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)*
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.)

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.

,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.

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
```

Index

Where more than one page reference is given, the major reference is first.

Indexes to systems reference library manuals are consolidated in the *OS/VS2 Master Index,* GC28-0663. For additional information about any subject in this index, refer to other publications listed for the same subject in the *Master Index.*

OS/VS2 Supervisor Services and Macro Instructions (S370-36) Printed in U.S.A. GC28-0683-1

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

OS/VS2 Supervisor Services and Macro Instructions

GC28-0683-1

READER'S COMMENT FORM

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. *For more direct handling of such requests, please contact your IBM representative or the IBM Branch Office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Index Figures Examples Legibility

Are the tables used to describe the macro instructions in this publication an improvement over the brackets and braces syntax?

What is your occupation? _____

Number of latest Technical Newsletter (if any) concerning this publication: _____

Please indicate in the space below if you wish a reply.

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.

Cut or Fold Along Line

Cut or Fold Along Line

Your comments, please . . .

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Fold Fold

First Class
Permit 81
Poughkeepsie
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department D58, Building 706-2
PO Box 390
Poughkeepsie, New York 12602

Fold Fold

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

OS/VS2 Supervisor Services and Macro Instructions (S370-36) Printed in U.S.A. GC28-0683-1

OS/VS2 Supervisor Services and Macro Instructions

GC28-0683-1

READER'S COMMENT FORM

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. *For more direct handling of such requests, please contact your IBM representative or the IBM Branch Office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Index Figures Examples Legibility

page 25, line 8: change "virutal" to "virtual"
page 27, Figure 8: align "CSECT" and "ENTRY" with "SAVE"
page 36, line 19: delete "copy."
page 37, line 21: change "is register 13" to "in register 13"
* page 38, line -7: change "The first" to "The second" [& change Figure 22?]
page 39, line 6: change "to register 0" to "in register 0"
page 50, line -11: change "seen" to "soon"
* page 196, lines -9,-10: if 'ECB=ecb addr' is specified, 'ecb addr' is the address of an event control block, not that of a fullword containing the address of an ecb. You have confused this with the list version. [Note that Example 1 doesn't really explain much either.]

* throughout: the description of the RELATED parameter should be given once in full, then with a pointer [e.g. "see page xyz"] in each macro allowing this parameter. If you must repeat it over and over, at least use the relevant macros each time instead of GETMAIN/FREEMAIN!

Are the tables used to describe the macro instructions in this publication an improvement over the brackets and braces syntax?

* No; I find the old notation [e.g. in GC28-6646] easier to understand; I can't tell what is optional or mutually exclusive with the new notation.

What is your occupation? Computer Science Researcher

Number of latest Technical Newsletter (if any) concerning this publication: None

Please indicate in the space below if you wish a reply. I would like a reply [to the asterisked items especially]. My name and address is:

Paul McJones
IBM Corp, Dept. K55 Building 282
San Jose, CA 95193

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. Elsewhere, an

Name | Internal Zip | City & State

To: Mr. Paul McJones

IBM Corp.
Dept. K55/Bldg. 282
San Jose

Date: October 18, 1976
ıe & Tie/Ext.: Alan R. Beebe
/Dept. Name: Information Development
p/City, State: D58/706-2/Poughkeepsie
mail address: P. O. Box 390

Subject: Your Reader's Comment

Reference:

Thank you for your reader's comment on OS/VS2 Supervisor Services and Macro Instructions, GC28-0683-1.

The errors that you found on pages 36, 37, 38, and 50 have been corrected in Technical Newsletter number GN28-2604, dated January 2, 1976. The other errors that you pointed out on pages 25, 27, 39, and 196 will be corrected in a future edition of the manual.

Your comment on the RELATED parameter reinforces a change that we have been considering. We are going to repeat the description of the parameter for each macro, but we will change the examples to use the appropriate macro each time.

Thank you for taking the time to write to us.

Alan R. Beebe

Alan R. Beebe

/ek